

2018

## Exploring New Paradigms for Mobile Edge Computing

Yutao Tang

College of William and Mary - Arts & Sciences, [kissingers800@gmail.com](mailto:kissingers800@gmail.com)

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

---

### Recommended Citation

Tang, Yutao, "Exploring New Paradigms for Mobile Edge Computing" (2018). *Dissertations, Theses, and Masters Projects*. William & Mary. Paper 1550154000.

<http://dx.doi.org/10.21220/s2-7mxc-ym51>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact [scholarworks@wm.edu](mailto:scholarworks@wm.edu).

# Exploring New Paradigms for Mobile Edge Computing

Yutao Tang

Guilin, Guangxi, China

Master of Engineer, Chinese Academy of Sciences, 2011

Bachelor of Science, Beijing University of Posts and Telecommunications, 2008

A Dissertation presented to the Graduate Faculty  
of The College of William & Mary in Candidacy for the Degree of  
Doctor of Philosophy

Department of Computer Science

College of William & Mary  
January, 2019




## APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of  
the requirements for the degree of

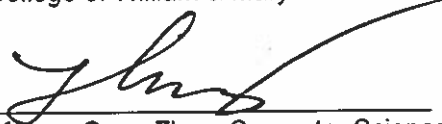
Doctor of Philosophy


  
\_\_\_\_\_  
Yutao Tang

Reviewed by the Committee, September 2018

  
\_\_\_\_\_  
Committee Chair  
Professor Qun Li, Computer Science  
College of William & Mary

  
\_\_\_\_\_  
Professor Weizhen Mao, Computer Science  
College of William & Mary

  
\_\_\_\_\_  
Associate Professor Gang Zhou, Computer Science  
College of William & Mary

  
\_\_\_\_\_  
Assitant Professor Adwait Nadkarni, Computer Science  
College of William & Mary

  
\_\_\_\_\_  
Dr. Ding Li,  
NEC Laboratories America Inc

# ABSTRACT

Edge computing has been rapidly growing in recent years to meet the surging demands from mobile apps and Internet of Things (IoT). Similar to the Cloud, edge computing provides computation, storage, data, and application services to the end-users. However, edge computing is usually deployed at the edge of the network, which can provide low-latency and high-bandwidth services for end devices.

So far, edge computing is still not widely adopted. One significant challenge is that the edge computing environment is usually heterogeneous, involving various operating systems and platforms, which complicates app development and maintenance. In this dissertation, we explore to combine edge computing with virtualization techniques to provide a homogeneous environment, where edge nodes and end devices run exactly the same operating system. We develop three systems based on the homogeneous edge computing environment to improve the **security** and **usability** of end-device applications.

First, we introduce vTrust, a new mobile Trusted Execution Environment (TEE), which offloads the general execution and storage of a mobile app to a nearby edge node and secures the I/O between the edge node 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 edge node, such that any sensitive data flowing through the hosted mobile OS is encrypted.

Second, we present MobiPlay, a record-and-replay tool for mobile app testing. By collaborating a mobile phone with an edge node, MobiPlay can effectively record and replay all types of input data on the mobile phone without modifying the mobile operating system. To do so, MobiPlay runs the to-be-tested application on the edge node under exactly the same environment as the mobile device and allows the tester to operate the application on a mobile device.

Last, we propose vRent, a new mechanism to leverage smartphone resources as edge node based on Xen virtualization and MiniOS. vRent aims to mitigate the shortage of available edge nodes. vRent enforces isolation and security by making the users' Android OSes as Guest OSes and rents the resources to a third-party in the form of MiniOSes.

## TABLE OF CONTENTS

Acknowledgments	vi
Dedications	vii
List of Tables	viii
List of Figures	ix
Chapter 1. Introduction	1
1.1 vTrust: Remotely Executing Mobile Apps Transparently With Local Un-trusted OS	3
1.1.1 Problem Statement	3
1.1.2 Contributions	5
1.2 MobiPlay: A Remote Execution Based Record-and-Replay Tool for Mobile apps	5
1.2.1 Problem Statement	6
1.2.2 Contributions	7
1.3 vRENT: Virtual Machine Migration on the Pervasive Edge for IoT apps	8
1.3.1 Problem Statement	8
1.3.2 Contributions	9
1.4 Overview	10
Chapter 2. vTrust: Remotely Executing Mobile Apps Transparently With Local Untrusted OS	11
2.1 Introduction	11
2.2 Overview	13

2.2.1	Design Goals	13
2.2.2	vTRUST Overview	14
2.2.3	Assumptions, Threat Model, and Scope	15
2.3	Detailed Design	17
2.3.1	Server Stub	17
2.3.2	Client Stub	17
2.3.3	Secure I/O Management	19
2.4	Implementation	23
2.4.1	Processing The Output	23
2.4.2	Processing The Input	25
2.5	Evaluation	27
2.5.1	Performance of Encryption/Decryption	29
2.5.2	Performance of Compression/Decompression	30
2.5.3	Throughput	31
2.5.4	Responsiveness	32
2.6	Security Analysis	34
2.7	Limitations and Future Work	35
2.8	Related Work	36
2.9	Chapter Summary	38
Chapter 3. MobiPlay: A Remote Execution Based Record-and-Replay Tool for Mo-		
	bile Applications	39
3.1	Introduction	39
3.2	Design of MobiPlay	41
3.2.1	Design Rationale	41
3.2.2	Architecture of MobiPlay	43
3.2.3	App Recording	45
3.2.4	App Replaying	46

3.3	Implementation	47
3.3.1	Physical Devices	47
3.3.2	The Client-Server Platform	48
3.3.2.1	SVMP Client	48
3.3.2.2	SVMP Virtual Machine	49
3.3.2.3	Networking	49
3.3.3	The Record Approach	50
3.3.3.1	MotionEvent	50
3.3.3.2	SensorEvent	51
3.3.3.3	KeyEvent	51
3.3.3.4	Location	52
3.3.3.5	Rotation	52
3.3.4	The Replay Approach	53
3.3.4.1	Replay on the Server	53
3.3.4.2	Replay on the mobile phone	54
3.3.4.3	Event Sampling	56
3.4	Evaluation	57
3.4.1	Usability	57
3.4.2	Latency	58
3.4.3	Time and Space Overhead	59
3.4.4	Event Sampling	60
3.5	Limitations and Future Work	61
3.6	Related Work	61
3.7	Chapter Summary	63
Chapter 4.	vRENT: Virtual Machine Migration on the Pervasive Edge for IoT Ap- plications	65
4.1	Introduction	65



4.2	Related Work	68
4.2.1	IoT Application	68
4.2.2	Smartphone in IoT	69
4.2.3	Edge computing	69
4.2.4	Virtualization and Migration	70
4.3	Problem and Overview	70
4.3.1	Preliminaries	71
4.3.2	Design Goals	71
4.3.3	vRENT Overview	72
4.3.4	Assumptions	73
4.4	Migration Procedure	75
4.4.1	What to Migrate	75
4.4.2	How to Migrate	78
4.4.2.1	How to design the migration process	79
4.4.2.2	Where to save Delta	79
4.4.2.3	How to locate Delta	79
4.5	Implementation	81
4.5.1	MiniOS Porting	82
4.5.2	Device APIs	82
4.5.3	Application Programming	84
4.5.4	Migration Implementation	84
4.6	Evaluation	86
4.6.1	Environment Setup	86
4.6.2	Boot Time	86
4.6.3	Save/Resume Time	87
4.6.4	Migration overhead	88
4.6.5	Impact Among Domains	88
4.7	Discussion and Limitations	89

4.7.1	Battery Drain and Wireless Charge	90
4.7.2	Incentives and Bootstrapping	90
4.8	Chapter Summary	91
Chapter 5.	Conclusion and Future Work	92
	Bibliography	95

## ACKNOWLEDGMENTS

This dissertation would not have been possible without the guidance, encouragements, patience, and support from my advisor, committee members, colleagues, friends, and family.

First I am exceptionally grateful to my doctoral advisor Qun Li. His patient guidance and constructive advice helped me obtain all my research outcomes. His enthusiasm in pursuing great research and rigorous academic attitude greatly impact my attitude towards research and even life. Also, without his kindly and unconditional support, it is impossible for me to finally go through my years in graduate school. It has been my greatest honor to have him as my advisor.

I would like to thank my dissertation committee, Prof. Weizhen Mao, Prof. Gang Zhou, Prof. Nadkarni Adwait, and Dr. Ding Li, for their wonderful insight, guidance, encouragements and time.

I am grateful to my colleagues and collaborators including Prof. Fengyuan Xu, Dr. Zhengrui Qin, Dr. Ed Novak, Zijiang Hao, Shanhe Yi, Chen Li, Yue Li, Dr. Zhichun Zhao, Dr. Mu Zhang and Dr. Zhiqiang Lin. I appreciate all the collaborations and helpful discussions that I had with them over the years.

I am also thankful to our Computer Science Department Chair, Professor Robert Michael Lewis, and the fantastic Computer Science administration team, Vanessa Godwin, Jacquelyn Johnson, and Dale Hayes, for their persistent support and help.

Finally, I want to give my gratitude to my family, whose love, support and encouragement are always with me in this journey.

I would like to dedicate this dissertation to my wife, Lingying Zhao, my son Gabriel Ruixuan Tang and my parents, Bicheng Tang and Jianwei Lv, who provided endless support and love throughout my time at William and Mary.

## LIST OF TABLES

2.1	The data structure embedded in each frame.	21
2.2	Comparison with the related work.	37
3.1	The devices used in MobiPlay system.	48
3.2	Details of data injection in white-box testing.	56
3.3	The apps that MobiPlay has recorded and replayed successfully.	57
3.4	The time and space overhead and number of events in each category.	59
4.1	The APIs for network, disk, GPS, and Bluetooth.	83
4.2	Boot time of MiniOS in milliseconds.	87

## LIST OF FIGURES

2.1	An Overview of vTRUST.	14
2.2	Screen Frames Processing in vTRUST.	24
2.3	Sensors Input Processing in vTRUST.	28
2.4	Comparison of different encryption algorithms.	29
2.5	Compression ratios.	30
2.6	FPS of vTRUST output.	31
2.7	vTRUST responsiveness.	32
3.1	App input data flow, within a mobile phone (no server).	42
3.2	MobiPlay consists of a mobile phone and a server.	43
3.3	MobiPlay records input data on both the mobile phone and the server.	45
3.4	MobiPlay can relay an app on both mobile phone and server, black-box and white-box testing, respectively.	46
3.5	The MotionEvent, SensorEvent and KeyEvent classes along with their associated fields.	51
3.6	The class LocationListener and its four methods with the corresponding parameters.	52
3.7	Round-trip time for different types of input.	58
3.8	Re-sampling reduces the replay time.	60
4.1	The scenario where the smartphone serves as the hub of IoT networks.	66
4.2	The migration procedure.	74
4.3	Virtual address space.	76
4.4	The layout of Disk0.	83

4.5	The migration implementation.	84
4.6	The save time and the resume time for different memory size.	87
4.7	Execution time of tasks in normal and migration cases.	88
4.8	The execution time for different number of extra MiniOSes.	89

# Chapter 1

## Introduction

Edge computing is becoming increasingly popular and is considered a future trend. Unlike Cloud data centers that are usually located in the center of the network, edge nodes are deployed at the edge of the network. As a result, edge computing allows data produced by end-devices, such as Internet of things (IoT) and mobile devices, to be processed closer to where it is created instead of forwarding it to a distant cloud.

Edge computing has many advantages. First, edge computing can significantly reduce the data volume that must be transferred to the cloud, thereby reducing transmission cost, decreasing latency and improving quality of service. Second, edge computing eliminates a core computing environment, avoiding major bottlenecks and potential single points of failure.

Bearing these features, edge computing has been applied to many scenarios, such as IoT, autonomous vehicles, smart cities, and industrial manufacturing, etc. Although edge computing has great potential, current projects mostly use edge nodes as a backup for more resource, while other aspects are seldom explored. One reason is that current edge Computing is composed of many heterogeneous edge devices, which may run different operating systems, such as Linux for PC, Android for the mobile device. Since each operating system has its own architecture and interfaces, the applications (apps in short) designed for one operating system cannot be directly implemented by others. Therefore, developers may have to redesign their products specifically for each edge computing use case, which significantly impedes the application of edge computing to areas other than performance and scalability.



In this dissertation, we aim to overcome this problem by bringing edge computing into a homogeneous environment, such that resources can easily be shared among the edge nodes and the end devices. Afterward, we explore various edge computing paradigms in this environment to provide better services to end devices. In a homogeneous environment, the end-devices and edge nodes run the same operating system, such as Android. Compared with a heterogeneous environment, it has many advantages. First, it allows the same app to run on both the end-device and the edge node without any modification. This feature makes edge computing compatible to legacy apps and thus significantly reduces the development cost. Second, it can provide users the same way to operate the apps as they used to do, without any learning cost or any harm to the user experience. Finally, it makes the maintenance much easier, since administrators only need to maintain one version of the apps for all platforms.

We integrate techniques, such as virtualization, to create a homogeneous environment to host edge computing paradigms. Under this setting, we design and implement three new systems to improve the **security** and **usability** of end-device apps with the assistance from edge nodes. First, we unveil the security and management issues of mobile devices, which could leak out sensitive data. We present a system that combines the virtualization and edge computing to provide a Trusted Execution Environment for sensitive mobile apps. Then, we systematically study state-of-the-art record and replay tools and analyze their pros and cons in mobile app testing. We find that no tool can collect all input data without modifying the Operating System. As such, we design and implement an edge computing based tool to record and replay all mobile input data, where the data is recorded and replayed on the edge node. Finally, we investigate edge apps in IoT and point out that the infrastructure insufficiency is the bottleneck that hinders the development of edge computing. We design a strategy that uses mobile devices as edge nodes to backup the IoT devices and implements a system that allows mobile users to securely share their spare resource in terms of MiniOS.

## 1.1 vTrust: Remotely Executing Mobile Apps Transparently With Local Untrusted OS

Mobile devices have become increasingly integral and ubiquitous in recent years, with over a billion active devices worldwide today [93], surpassing desktop computers as the most popular personal computing platform [11]. Inevitably, this trend has made many organizations 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 [21]. For instance, hospitals have allowed doctors and nurses to access patient healthcare records using mobile devices, and government agencies and militaries have allowed classified documents accessed and processed with mobile device [2–4].

### 1.1.1 Problem Statement

Unfortunately, along with the convenience, the use of mobile device also brings unprecedented management and security challenges, especially for security and privacy sensitive apps. On the management side, while mobile devices used for work are part of the organization’s network, ensuring that all mobile devices are complying with the security policy is very challenging [28] for a variety of reasons. First, many work phones are misused for personal purposes since carrying both a personal and a work phone can be painful for users [7, 17]. Second, there usually lacks a comprehensive measure to guarantee that the user is following a secure practice when handling sensitive data. For example, a user may download classified documents and then upload them to a public cloud for easier editing, even though the security policy is against such a conduct. Third, the IT department may revise their security policies from time to time to cope with the newest security update [100], which makes the problem even worse.

On the security side, new forms of malware targeting mobile devices are on the rise with the increasing popularity of the mobile devices. For instance, mobile malware increased more than three times between 2015 and 2016 [26]. 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 [27] to advanced persistent threats (APT) [5], can persistently and stealthily steal valuable data from the mobile device. Even for security sensitive department, such as the military, may fail to properly protect the devices. For example, most recent Israeli military personals were reported being spied by attacks through trojanized apps in mobile devices [19].

A variety of approaches have been proposed to protect sensitive data on a mobile device. One widely-used approach is the full-disk encryption [1, 59], aiming at securing the mobile storage. Several advanced schemes use a remote trusted server to store the decryption keys [66, 145] 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 the sensitive data.

Other works [35, 129, 130] leverage TrustZone [10] to create a trusted execution environment (TEE) for security sensitive apps. In particular, TrustZone provides two physically isolated execution environments. The normal world runs rich OSes for untrusted apps, and the secure world runs a trusted minimal OS for security-critical code (SCC) [96, 108, 109]. All SCC must be carefully verified to prevent vulnerabilities before deploying in the secure world since the secure world is fully trusted and has the highest privilege; otherwise, any glitch could lead to a system-wide compromise [49]. Further, hosting too much SCC in the secure world may result in breaking the security. Because the bugs and vulnerabilities are usually proportional to the size of the code, no matter how carefully they are designed and examined [114]. Finally, TrustZone-based solutions often require redeveloping the app (legacy unfriendly), making it harder to reuse (and execute) any existing apps.

In order to tackle both the security and management challenges, a promising direction is to offload security sensitive apps to a server [2], such as using Virtual Network Computing (VNC) [15] or Secure Virtual Mobile Platform (SVMP) [8]. This scheme sandboxes the execution and storage of sensitive apps in a verified remote execution environment, and uses a proxy to transmit I/O data (e.g., sensing, touchscreen input, and screen output) at the mobile OS in a transparent manner. 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 the sensitive data.

Unfortunately, a serious drawback for the existing remote execution solutions is that they do not take the I/O protection into consideration. 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 [101] or sensors [45, 113, 120, 153]. More sensitive data can even be derived, such as deriving passwords from personal information [97, 98, 147].

### 1.1.2 Contributions

In this dissertation, we introduce a new TEE for mobile apps based on a hypervisor on the mobile device and a virtualized mobile OS running on an edge node. Similar to existing remote execution solutions, our solution enforces data security by outsourcing the computation and storage of a sensitive app into a security-enhanced virtual machine (VM) running on an edge node (e.g., a VM managed by an enterprise). In addition, our solution leverages the hypervisor on the mobile device to protect I/O data by establishing an encrypted I/O channel between the edge node and the local hypervisor. Specifically, all input data is encrypted in the hypervisor before entering the local mobile OS and decrypted in the edge node. The output data works similarly in an opposite direction. In this way, the local mobile I/O is protected from unauthorized access from local mobile OS. Meanwhile, our solution allows users to install non-sensitive apps on the local mobile device and provides a mechanism to seamlessly switch between (non-sensitive) local apps and (sensitive) remote apps.

## 1.2 MobiPlay: A Remote Execution Based Record-and-Replay Tool for Mobile apps

Mobile devices have been increasingly popular in recent years with nearly two billion users world-wide [57], and millions of apps (apps) are available in each of the substantial platforms. As new

technologies, such as various sensors, and other rich resources are adopted by mobile devices, the user experience is greatly enhanced. However, at the same time, these new features have imposed challenges on app design and testing for developers. Nowadays, running an app may involve multiple input sources: touchscreens (swiping, pinching, zooming, click/tapping), sensors (GPS, accelerometer, compass, gyroscope), and networking (online gaming, websites, Bluetooth) to name a few. Therefore, it is challenging for developers to test and debug mobile apps, since it is non-trivial to accurately record the data from all these inputs as well as the interaction among different components involved in the app. Even after the input has been recorded, it is then challenging to replay the app execution using the recorded data. While a handful of record-and-replay tools have been developed for mobile apps, they are far from perfect.

### **1.2.1 Problem Statement**

As one of the key technologies in software engineering, the record-and-replay approach to software testing has played an important role in the development of mobile apps. Record-and-replay is a necessary and valuable tool for mobile app development because it allows developers to easily find and recreate elusive and complex bugs, test outlier cases, and increase the speed of testing software by automating the process. Record-and-replay improves the software in the testing, debugging, optimization, and upgrading phases. However, we face several challenges in implementing such a system on mobile devices. Considering the rich input capabilities of mobile devices and the real-time interaction between the mobile app and the user, the challenges are as follows. First, it is difficult to accurately record an app's continuous execution instead of some discrete actions. Second, it is hard to record all the input data, which is especially true for sensors such as the GPS. Third, it is preferable that all recorded data is human-readable, such that developers can easily analyze, revise, and re-assemble the recorded data in order to accurately locate and identify bugs or performance bottlenecks. Finally, it may be possible to modify the mobile devices operating system (OS) to achieve record-and-replay functionality. However, modifying the OS requires a device with an unlocked bootloader and an open source operating system. Unlocking the bootloader is impossible on some devices (due to manufacturer obstacles), difficult, and usually erases all user data on the

device. Modifying the operating system may introduce bugs, and is difficult in general, requiring access to any proprietary closed-source components from the original.

While researchers have made numerous efforts to develop and improve the record and replay tools for the mobile device in recent years, the results are frustrating, because none of them can truly capture all possible input data. The reason is that the current mobile system such as Android does not provide corresponding APIs or privilege for record tool to collect data at the User level. Moreover, the mobile device vendors are reluctantly let the users even developers to get the root privilege of their Mobile OS for security concerns. This can further limit the functionalities of record and replay tools.

### **1.2.2 Contributions**

In this dissertation, we design a tool to record and replay an Android app’s execution by creating a homogeneous edge computing environment. The to-be-tested app is actually running on the edge node which provides the same runtime environment as the local mobile device. Meanwhile, there is a proxy app running on the local mobile device at the User level. During the recording stage, the proxy app forwards all input data such as touch screen data and sensor data to the edge node. In the meantime, It receives the GUI update of the to-be-tested app from edge node and renders it on the local mobile device as if the app were running locally. Our tools have many advantages. First, it does not need the root privilege of the mobile device since the proxy runs at the User level. Second, it is able to record all sensor data inputs, for replay later, in the form of high-level events, such as touchscreen gestures, the key event, and sensor event. Third, Besides solving the existing problems and challenges we outlined previously, our tool is also able to offer more flexibility than ever before. It can not only record all input data, on both the mobile device and the server side but it can also replay the app on both sides as well. Finally, our tool is suitable for both white-box testing and black-box testing.

## 1.3 vRENT: Virtual Machine Migration on the Pervasive Edge for IoT apps

Internet of Things (IoT) has emerged as the future of the Internet, integrating rapid developing technologies such as mobile computing, wearable computing, wireless networks, and cloud computing. It has already made an enormous impact on business, society and human daily life by a series of innovating products in fields of the smart grid, smart vehicle, smart home, smart city, connected health, AR/VR, etc. As endpoint of connected smart objects, IoT benefits a wide range of apps (e.g., surveillance, environment control, intrusion detection, traffic management, wildfire watch, etc.) with the deployment of massive smart devices at various places.

### 1.3.1 Problem Statement

Though IoT brings many benefits, it is far from being perfect. One big issue of the current IoT is that those devices are mostly resource-limited in collecting, on-site processing and sharing a large volume of data for advanced analytics. One way to mitigate this problem is to back IoT systems with cloud services, but it cannot catch up with the pace of the increasing demands for cost efficiency, low latency, scalability, resource utilization, mobility support, and location-awareness. As a result, edge computing (a.k.a fog [39], cloudlet [133]) is recently proposed to push computations from cloud to the edge of networks, where deploying extra cloud-scale data center may be prohibitive in terms of cost [132]. In edge computing, the device that shares its resources with nearby clients is called the *edge node*. While there is no restriction on what type of devices can be an edge node, most existing literature or proposals prefer resource-rich edge nodes, like servers, high-end desktops or laptops, since they not only have sufficient resources but also support virtualization techniques to provide isolation. However, deploying those devices usually means extra cost, low mobility, which are all important obstructive factors considered by end users.

### 1.3.2 Contributions

In this dissertation, we explore the feasibility and techniques that can turn mobile devices into lightweight edge nodes to enhance current IoT systems. In addressing these challenges, we develop a new system to manage device resource of mobile device based on Xen virtualization and MiniOS. This system enforces isolation and security by elevating user's Android OS as a Guest OS and renting mobile device's resource in the form of MiniOS. Specifically, in our system, the hypervisor and Dom0 manages all resources. The mobile device users and renters can only access the resources in the Guest OS and MiniOS respectively, they cannot access any resource beyond their domains. In addition, our system presents an effective and efficient homogeneous scheme for live MiniOS migration, which allows unfinished tasks running in the MiniOS to be migrated to other entities when needed.

The benefits provided by our system are prominent. In term of security, it sandboxes the MiniOS and Guest OS in two different domains and prevents them from accessing the resource of each other. Thus, the mobile device owners do not need to worry about their software/hardware being harmed by renting their idle resource, and renters are willing to run high assurance tasks on the rented resource. Furthermore, our system can easily manage the rented resource. By renting resource in the form of MiniOS and supporting live migration, renters are more flexible for designing their apps. They can manage and access all resource of MiniOS, and decide when to start, stop and migrate the tasks as they want.

Comparing to traditional migration methods [20, 40, 41, 54, 87, 131, 133], our system is more flexible and efficient. Traditional migration methods rely on the hypervisor to save system states. During migration, traditional methods usually require hypervisor to over-conservatively collect all system states and user data, leading to massive image volume. Even though they have been proved useful for PCs and cloud servers since they have more resources, they are inefficient in our renting scenarios. On the contrary, our system allows the MiniOS to save system states, giving MiniOS more freedom since it can decide when to migrate without fully shutting down the MiniOS. Meanwhile, our system can smartly locate the useful data that must be migrated, usually generating far less data than traditional migration methods.



## 1.4 Overview

The remaining dissertation is structured as follows. chapter 2 presents a TEE for high assurance mobile apps. chapter 3 introduces a novel approach to record and replay device I/O data without requiring any extra privilege from the mobile device. chapter 4 presents a new methodology that allows users to securely share their spare mobile device resource to third-party for IoT data processing. Finally, chapter 5 concludes this dissertation.

## Chapter 2

# vTrust: Remotely Executing Mobile Apps Transparently With Local Untrusted OS

### 2.1 Introduction

In this chapter, we present vTRUST, a new TEE for mobile apps based on a hypervisor on the mobile device and a virtualized mobile OS running in a remote server. Similar to 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 hypervisor on the mobile device to protect I/O data by establishing an encrypted I/O channel between the remote server and the local hypervisor. Specifically, all input data is encrypted in the hypervisor before entering the local mobile OS, and decrypted in the remote server. The output data works similarly in an opposite direction. In this way, the local mobile I/O is protected 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. By virtualizing the execution environment on a server, it also makes powerful but resource-hungry security enhancements, such as anti-virus, VM-introspection, and data flow tracking techniques, feasible to deploy. Meanwhile, it is also resilient to device losses since the data is not stored locally at all. Furthermore, vTRUST also makes management much simpler. All VMs on the server are within the same network and are fully controlled by the IT department of an organization. The access of data can be granted or revoked at any time, without concerns of physical presence of the device or the network availability. The IT department can also easily track the flow of the data to ensure they are handled properly. Additionally, it provides a manageable way to upgrade the system and apply system patches. For example, any update on the security policies can be immediately applied to the VMs.

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 the performance of 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 the 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 both easier management and stronger protection by executing sensitive mobile apps in a trusted remote server with secure I/O protected by local hypervisor.
- We have implemented a prototype of vTRUST on an Arndale development board and a server running Android-X86. We have also developed a number of optimizations to enhance the performance of vTRUST.
- We have evaluated the performance of vTRUST and observed that vTRUST incurs little overhead. In addition, there is little user experience change thanks to the transparency offered by vTRUST.

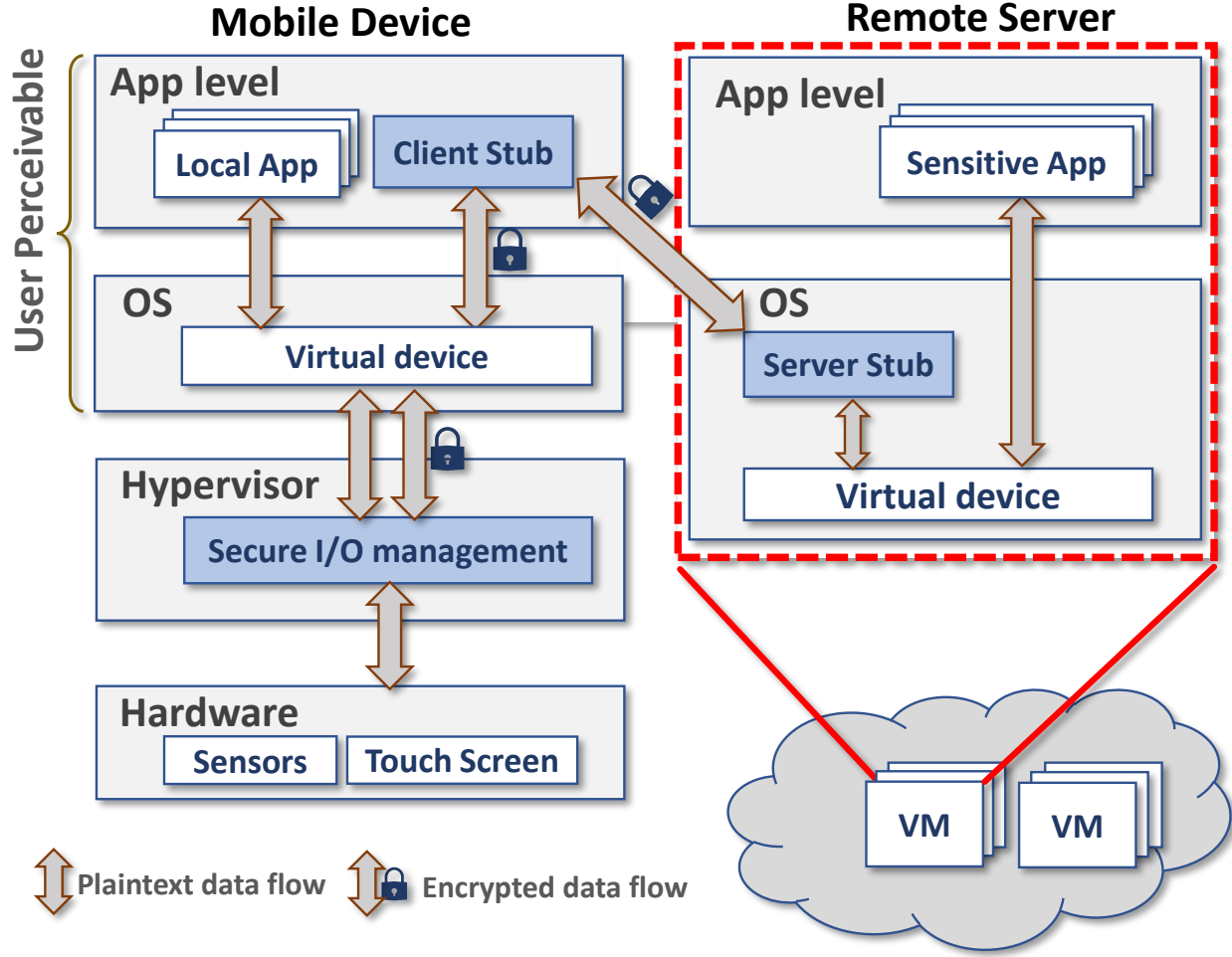
## 2.2 Overview

In this section, we provide an overview of vTRUST. We first describe our design goals in §2.2.1, then describe how vTRUST works at §2.2.2, and finally present the threat model, scope, and assumptions in §2.2.3.

### 2.2.1 Design Goals

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

- **Full Protection.** A compromised mobile OS must not be able to 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 must not require any modification on the mobile apps, such that legacy apps can still be executed in our system.
- **Easier Management.** vTRUST should enable an IT administrator to leverage any existing security measures to protect the sensitive app. It should also make the system and app update easier.
- **Seamless Switching.** vTRUST must concurrently support both protected apps running on a server and unprotected apps running in the untrusted mobile device. It must ensure a seamless switch at the mobile client between protected and non-protected apps.
- **Unchanged User Experience.** vTRUST also needs to keep the user experience unchanged to ensure usability. Users are able to operate the sensitive apps as if they are installed on the local mobile device.
- **Low Overhead.** The protected apps must function properly with an acceptable performance overhead.



**Figure 2.1:** An Overview of vTRUST.

### 2.2.2 vTRUST Overview

An overview of vTRUST is presented in Figure 2.1. At a high level, it uses a client-server architecture and consists of three key components: Secure I/O Management and Client Stub running in the client mobile device, and Server Stub running in the remote server (e.g., a VM). To execute a security sensitive app in vTRUST, essentially end users just launch and interact with the app, which is executed in the remote server, from a local mobile device through a secure communication channel protected by the Secure I/O Management component located in the local hypervisor. Any input from the mobile device (e.g., sensors, and touch positions) will be encrypted by the hypervisor, and then delivered to the local mobile OS, and further delivered to the remote server by the Client Stub,

which communicates with the Server Stub. Then, the Server Stub decrypts the data and provides to the apps. If the sensitive app has any output, it will correspondingly go through the same channel (i.e., from the Server Stub, to the Client Stub, and then decrypted at the hypervisor). Therefore, an untrusted mobile OS is not able to view any of the data of the sensitive apps, which are executed in the trusted remote server.

A key enabling technique of vTRUST is the virtualization. Both the local mobile device and the remote server leverage virtualizations for different purposes. The mobile device uses it to protect I/O data from 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 dissertation, we use the “Hypervisor” to refer the hypervisor on the mobile device, the “VM” to refer the VM on the server, and the “mobile OS” to refer the local mobile OS.

Also note that the I/O communication channel between the local hypervisor and remote VMs is protected by symmetric encryption. The keys are generated and distributed by the IT administrator through pre-installation or a portable device (e.g., an SD card), and are stored in the hypervisor’s storage, which is also physically isolated from the mobile OS.

### 2.2.3 Assumptions, Threat Model, and Scope

**Assumptions.** We consider a trusted computing base (TCB) that includes (1) the hypervisor at the client side, and (2) 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 of deploying the hypervisor on the mobile device. Our assumption is practical since the hypervisor comes with the IT department, does not require any third-party code, and cannot be modified by the end users. This assumption is also widely adopted by many efforts (e.g., [48, 50]).

The remote VMs are assigned to the users, and are maintained by the IT administrators. Each VM is carefully protected and monitored by applying advanced techniques, such as anti-virus softwares, fine-grained system log systems, firewalls, and intrusion detections. We also assume the apps installed on the VMs are trusted. To securely access sensitive data, these apps should be carefully developed and verified. The IT administrator installs apps and possible tools on the VMs for the

users and disallows customized installation of any un-trusted apps. This makes the VMs much less likely to be infected by malware and alike.

**Threat Model** The client mobile OS is not trusted. Users are free to install any application on their own mobile OS. Typically, the mobile OS may have many third-party applications and libraries installed, exposing the mobile OS to various attack vectors, such as rootkits, Trojans, Keyloggers, or capturing screenshots. Therefore, an attacker can potentially take over the mobile OS by penetrating its large attack surface. As a result, the attacker can access all the resources that are available to the mobile OS. Furthermore, she can eavesdrop on and manipulate any hop of the network connection between the mobile device and the server.

**Scope** With respect to the scope, we focus on the platform running Android apps. In particular, on the server side, we use the Android for x86 [81] as the VM to host sensitive apps. We choose x86 other than ARM as our server platform for three reasons. First, large enterprise servers are commonly x86-based. Second, most Android apps are written in Java language, which can be implemented on both x86 and ARM platforms. with third-party native libraries will more likely provide their Android for x86 version in the future.

On the mobile device side, we focus on the off-the-shelf ARM platform powered by Android mobile OS. In order to provide a virtualization environment, we leverage KVM/ARM [58] as the underlying hypervisor. In the KVM/ARM system, KVM is implemented in the host OS as a kernel module, and it utilizes the ARM virtualization extensions to provide fully virtualized CPU and memory. Meanwhile, KVM/ARM leverages QEMU and Virtio [127] user space device emulation to provide I/O virtualization [58]. QEMU is one of the several user level processes that run in the host OS, providing virtual hardware devices, such as the touch screen and sensors to the guest OS. VTRUST performs I/O data encryption and decryption in QEMU since these virtual devices bridge the mobile OS and the hypervisor, and encrypting/decrypting the data there causes the least impact on other modules.

## 2.3 Detailed Design

In this section, we present the detailed design of vTRUST. We first describe how we design our Server Stub in §2.3.1, then Client Stub in §2.3.2, and finally Secure I/O Management in §2.3.3.

### 2.3.1 Server Stub

The server in vTRUST is in charge of running sensitive apps. The server is anticipated to be protected by powerful security infrastructures, such as anti-virus software, intrusion detection, etc. It can host many VMs for multiple mobile devices, atop of each VM runs an Android mobile OS. For clearer presentation, we only show a single VM instance in 2.1. 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.

A Server Stub is installed in the Android Framework of each of the VM to tunnel the communication between the client mobile device and the sensitive apps. Specifically, it receives input data such as touchscreen or sensor readings from the local device and delivers them to the applications, as well as receives output data, such as audio or screen frames, from the apps and delivers them to the mobile device. However, as the sensitive apps have no awareness of the server stub’s existence, they only communicate with the underlying Android framework for I/O like what they normally do. To make the communication transparent, we virtualize the hardware and feed the data from the Server Stub to these virtualized hardware, such that the above-lying apps can consume data from the mobile device transparently, as if they were generated locally.

### 2.3.2 Client Stub

The mobile device runs a hypervisor, atop which a single mobile OS is hosted as the guest OS. Note that the overhead of virtualization on mobile device is actually very light. For instance, it has also been demonstrated that KVM on ARM achieves a performance being near native [143]. vTRUST leverages the hypervisor to govern all of the I/O generated from or sending to the hardware, and act as a medium between the mobile OS and the actual hardware.



An end user views the hypervisor as completely invisible since it has no user interface, and the mobile OS occupies the whole screen. The hypervisor only performs encryption and decryption for I/O of security sensitive apps. For instance, the encrypted screen frames from the server can be properly rendered on the physical device after decrypting the content in the hypervisor. Similarly, the user input, such as the touchscreen data, is encrypted in the hypervisor and then delivered to the mobile OS, which is then further delivered to the remote server. The keys for encryption/decryption are managed by the hypervisor and this part of storage is never made accessible from the mobile OS. As such, the I/O data is indecipherable from the perspective of the mobile OS. In case that the mobile OS is compromised, the sensitive applications can still be safely executed on the remote server.

In vTRUST, end users only interact with the mobile OS. The mobile OS is fully functional such that the user is free to install any apps locally and uses it as a normal mobile device in exactly the same way as before. Providing this feature in vTRUST is critical for usability consideration, since the user is also able to use the device for any personal purposes. Regarding the sensitive apps, their (encrypted) I/O data are proxied to the Server Stub of the remote VM through a Client Stub, which is a normal mobile app and acts as a portal connecting to the remote server. Launching the Client Stub establishes a network connection to the VM on the server (the corresponding Server Stub), and the screen is thereafter switched to show the decrypted frames from the VM, just like a VNC or remote desktop app. The decryption is performed in the hypervisor as explained before. Thus, a user is indeed viewing the desktop in the server VM at the Client stub. Meanwhile, the Client Stub receives local input data and tube them to the VM. As such, a user is able to manage apps on the VM through interactions from the local device. On the other hand, to switch back to the local mobile OS, the user simply presses the "Home" button on the mobile device.

We choose to use a Client stub running on the mobile OS as the I/O relay mainly for the following reasons. First, we aim to provide vTRUST with a minimum user experience change. Having a Client Stub enables users to manage sensitive apps on the remote server in the same way as they are local. In contrast, if using hypervisor to communicate directly to the remote side, the system complexity of hypervisor will be largely increased, and the system overhead will also become larger due to the

frequent switches between the hypervisor and the mobile OS. The way of using the Client Stub is more suitable to our design goal and maintains a satisfactory user experience. Second, opening network connections may enlarge the attack surface, leaving the hypervisor susceptible to probing or various attacks from the network. Third, the implementation is cleaner. In our design, the hypervisor only needs to handle encryption/decryption and provide a virtual environment for the guest OS. It does not have any interface for the outside world. Otherwise, it would be highly error-prone and make vTRUST hard to maintain and upgrade due to heavy customization. Moreover, making the hypervisor directly communicate with the server demands the implementation of many auxiliary modules, for example, a UI system, a garbage collection and recovery mechanisms when an application crashes, etc.

### 2.3.3 Secure I/O Management

vTRUST builds a secure communication channel between the server and the mobile OS through the cooperation between the Server Stub and the Client Stub. The communication is done through the standard TCP protocol. The security of the data is ensured by vTRUST’s encryption mechanism. Thus, there is no additional protection required on the communication channel.

However, there are still several issues remained to be solved. For example, the hypervisor manages I/O for both remote sensitive apps and local non-sensitive apps. There must be a way for it to know whether the data is for sensitive apps or non-sensitive apps, such that encryption/decryption can be applied appropriately. To this end, there must be a channel to convey control signals. While we could use introspection techniques to infer the execution state of sensitive apps and non-sensitive apps from the hypervisor, such an approach would be too heavy.

Instead, we leverage a one-way communication channel from the remote server to pass the state information to the hypervisor in vTRUST. Specifically, vTRUST embeds control data in each output frame from the remote server. Some of these control data are encrypted just like the screen frame data and others are not. Note that this control channel is only one-way, from the Server Stub to the hypervisor. We use such a one-way communication only when the mobile device needs to receive commands from the server. There are three pieces of essential information that are included in the

control data and we explain each of them in the following.

- (I) **Differentiating data source.** vTRUST allows the user to run both local apps and remote sensitive apps simultaneously, and the screen frames are treated differently. Namely, the frames from remote apps should be decrypted before being passed to the physical device driver of the screen. However, from the perspective of the hypervisor, it is unclear whether a new frame is from a sensitive app or from a non-sensitive app. Furthermore, vTRUST features seamless switching between the two types of apps. For example, when the guest OS switches non-sensitive apps to the background and brings the client stub app (and thus the sensitive app) to the foreground, the hypervisor should begin decrypting the frames. The opposite process can also happen at any time. In essence, encrypted and non-encrypted frames may interleave with each other. Therefore, it is necessary for the frame to be self-contained.

To this end, we use a mark to indicate that a frame is from the remote server. This mark should not be encrypted as itself is an indicator of whether the data is encrypted. Whenever a mark is present, vTRUST enters the security mode we call Shield Mode, which means that the output is encrypted in the server stub and is supposed to be decrypted in the hypervisor, such that the sensitive data in the frame is kept safe from the untrusted mobile OS.

- (II) **Determining Encryption/Decryption Keys.** In vTRUST, the mobile device can host multiple Client Stubs, and thus they can communicate to multiple Server Stubs on the remote VMs. It is important to keep them isolated from each other. This isolation necessitates the use and management of multiple pairs of encryption/decryption keys since different VMs need different keys.

Therefore, the VM and the hypervisor need to synchronize the key pair that is used. The VM informs the hypervisor which key to use for decryption through a control signal embedded in the screen frame. Meanwhile, if input data is needed for the sensitive apps, another piece of control data is included, specifying the key used for encrypting the input.

- (III) **Selective sensor data transmission.** In Shield Mode, the input data is encrypted by the hypervisor, such that the guest OS can only see encrypted data. However, sensor data is

shared among all apps requesting it, implying that even though remote apps work properly in Shield Mode, local apps are receiving encrypted data without the awareness. Hence, they are likely to generate false results as a side-effect. For example, when both sensitive and non-sensitive apps are using the accelerometer data, sensitive apps will receive correct data. However, non-sensitive apps, such as a pedometer, may not function well due to receiving encrypted data.

To address this effect, we choose to only encrypt and transmit the input data that is required by the sensitive apps. vTRUST achieves this by including another piece of control data that specifies which sensor data should be encrypted and sent to the server. Such a selective data transmission scheme also helps to reduce the network overhead.

**Protocol Specification** In order to integrate control data in the output frames, we use four pixels in the bottom right corner of the screen and use them to encode an eight-byte data structure. When the Client Stub receives a frame, it will send it to the virtual screen, which is managed by the hypervisor. By reading the four pixel in the received frame, the hypervisor can easily extract the control data and process the input/output data accordingly.

Note that each encrypted frame loses four pixels. Fortunately, the bottom of the screen is typically reserved by Android for the navigation bar, and changing only four pixels will not greatly impact user experience. The four pixels embeds an eight-byte data structure, as shown in Table 2.1. In particular, this data structure consists of four fields.

Field	Size (bytes)	Description
MARK	2	0x55aa
DECRYPTION_ID	1	Frame Decryption Key
ENCRYPTION_ID	1	Input Encryption Key
SENSORS	4	Bit 0 - Accelerometer Bit 1 - Gyroscope Bit 2 - Magnetic Field Bit 3 - Gravity Bit 4 - Orientation Bit 5 - Rotation Vector ...

**Table 2.1:** The data structure embedded in each frame.

- (i) The MARK field takes two bytes, storing a constant value 0x55aa to notify the hypervisor to enter the Shield Mode. Note that it is possible to have false positives for this constant value, but this false positive can only occur in non-sensitive apps and the hypervisor decrypts the frame when it is not supposed to. Therefore, in the worst case it merely causes a non-sensitive app frame being unreadable, and it does not leak any sensitive data. Furthermore, this false positive is very unlikely to happen, and we have never observed any such incident.
- (ii) The DECRYPTION\_ID field takes one byte, informing the hypervisor the encryption key used on the current frame, such that the hypervisor can decrypt it with the corresponding key.

Note that MARK and DECRYPTION\_ID should not be encrypted since they deliver essential information on whether a decryption should be done or how to do the decryption. In contrast, the following two fields should be encrypted.

- (iii) The ENCRYPTION\_ID field takes one byte, informing the hypervisor which encryption key should be applied to the input data.
- (iv) The SENSORS field takes four bytes (i.e., 32 bits) and each bit in this field represents a sensor device, as shown in Table 2.1. Setting a bit to “1” indicates the data from the corresponding sensor is requested by the sensitive app and the data will be forwarded to the Server Stub, while setting a bit to “0” indicates the data is not used by the sensitive app and should be directly sent to the guest OS as is. For example, SENSORS = 0x03 means that the input data from the accelerometer and the gyroscope are used by the sensitive app and will be encrypted before forwarding to the guest OS, while data from the other sensors will be delivered to the guest OS in the plaintext form. Note that this field only contains sensor data; other data, such as that from touchscreen, will always be encrypted in Shield Mode.

This data structure encodes all the control data we have introduced. We present how an output frame is processed by the hypervisor in algorithm 1 and how the input data is generated to mobile OS in algorithm 2, with assistance of annotations introduced before. The detailed implementation of how we process the output frame, and generate the input data is presented in §2.4.

---

**ALGORITHM 1:** Output Frame Processing

---

```
frameBuffer  $\leftarrow$  getVirtualScreenFrameBuffer()
mark  $\leftarrow$  getMark(frameBuffer)
if mark == 0x55aa then
    shieldMode  $\leftarrow$  TRUE
    DECRYPTION_ID  $\leftarrow$  getFrameDecryptionMethod(frameBuffer)
    frameBuffer  $\leftarrow$  decryptFrame(frameBuffer, DECRYPTION_ID)
    (ENCRYPTION_ID, SENSORS)  $\leftarrow$  getDataStructureInfo(frameBuffer)
end
else
    | shieldMode  $\leftarrow$  FALSE
end
sendFrameToPhysicalScreen(frameBuffer)
```

---

---

**ALGORITHM 2:** Input Data Generation.

---

```
(inputData, inputType)  $\leftarrow$  getInputDataInfo()
if shieldMode == TRUE and SENSORS[inputType] == 1 then
    | inputData  $\leftarrow$  encryptInputData(inputData, inputType, ENCRYPTION_ID)
end
SendInputDataToGuestOS(inputData)
```

---

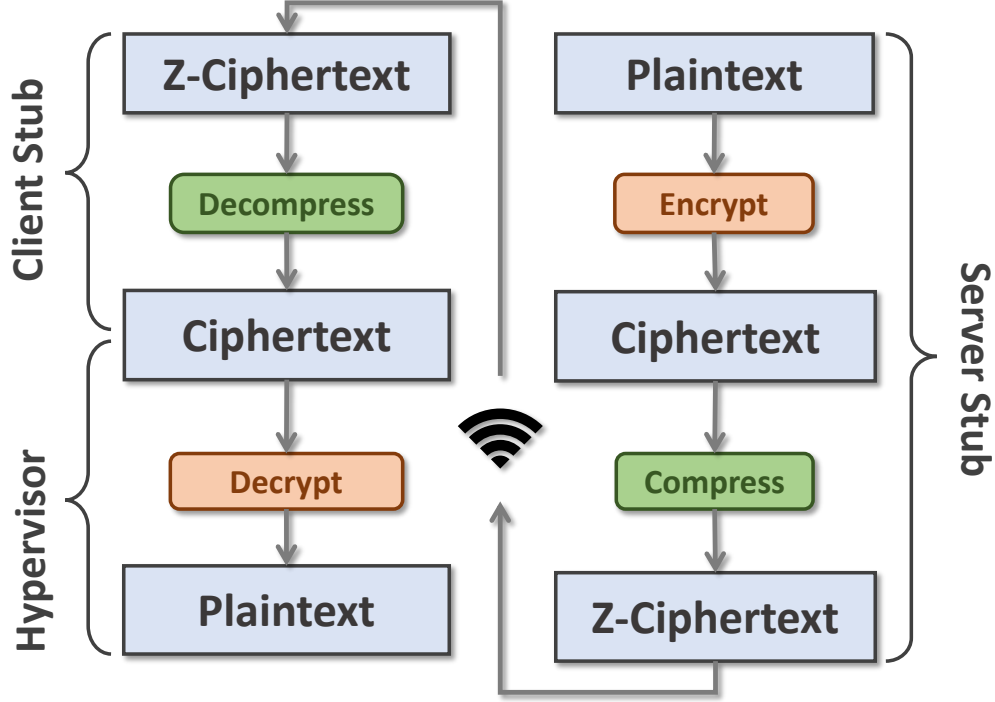
## 2.4 Implementation

In this section, we share the implementation details of vTRUST. Most of our implementation lies in how we transparently handle the I/O for the sensitive apps at both the server and the mobile device. Therefore, we first describe how we handle the output in §2.4.1, and then the input in §2.4.2.

### 2.4.1 Processing The Output

Typical output of mobile apps are screen frames [95] and audios. Currently, we only implement the screen frame output for its prevalence since almost all apps desire screen display. We note that the audio output can follow a similar procedure as screen frames.

The process of how screen frames are transmitted in vTRUST is illustrated in Figure 2.2. In a typical Android system, all screen update will eventually be sent to the framebuffer (i.e., `/dev/-graphics/fb0`) for screen rendering. As such, the Server Stub first fetches plaintext frames from the frame buffer, which is located in the OS kernel (that also explains why vTRUST is transparent to the execution of mobile apps), and then compresses the received frame data, the result of which is



**Figure 2.2:** Screen Frames Processing in vTRUST.

denoted as Z-ciphertext. The Z-ciphertext is sent to the Client Stub on the mobile device through the network. Afterwards, the Client Stub decompresses the data and sends the recovered ciphertext to the hypervisor, and the hypervisor decrypts the data with the corresponding decryption key specified in the frame. When implementing the frame data transmission channel, several critical issues need to be addressed. We elaborate each of them in the following.

- **Frame Data Integrity.** vTRUST must ensure the frame data received by the hypervisor is decryptable. In practice, we notice that the Android system will adjust the resolution of the screen data to fit in the local screen. In this case, the encrypted screen data will be distorted, and thus become not decryptable. To solve this problem, the Server Stub will adjust the resolution of the VM's virtual screen when it receives a connection request from the Client Stub, such that the size of the frame data matches the screen of the mobile device. Meanwhile, the Client Stub should also run in full-screen mode to avoid the data being further adjusted. This is usually the case since Android is designed to run a single app on the foreground, meaning that the app occupies the entire screen.

- **Encryption Algorithm.** In our implementation, we use the AES-128 block cipher [141] with CBC mode. A main reason is that AES is a mature and widely-adopted encryption technique, which is generally considered secure. Besides, according to our evaluation in §2.5.1, AES is the most efficient encryption algorithms in our evaluation. Note that we divide the frame into segments that contain 10 blocks, and apply AES-128 on each of the segment with a unique encryption key. In this way, we can enhance the security of the encrypted data and support more efficient compression, which is presented in the following.
- **Data Compression.** Sending frames usually involve heavy network overhead. Suppose QEMU provides a 640x480 virtual screen with RGB-565-color encoding to the guest OS, each frame would contain 614,400 bytes of pixels. To provide a decent frame rate (more than 15 frames per second) even under non-ideal network conditions, reducing the transmission data size of vTRUST is important. Fortunately, after analyzing the frames, we have found that many apps do not change their GUI significantly. As such, we design a simple compression algorithm, which calculates the difference between two neighboring frames using exclusive-or operation, and then we use LZ4 [13] algorithm to compress the outcome to maximumly reduce the data load. Note that we segmentize the frame and use AES encryption on each of the segment, so altering a value will only affect the current segment. Thus, the delta of 2 encrypted frames is largely preserved.

#### 2.4.2 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 is widely used in today’s mobile device and can play both the roles of the mouse and the keyboard of a desktop. The touchscreen data 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, the hypervisor will encrypt the touchscreen data from the hardware, and then feeds the encrypted data to the virtual devices supporting the local mobile OS. The ciphertext will be encapsulated in an Android-specific event (i.e., MotionEvent), consumed by the



Client Stub, and sent 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`).

Similar to the screen frames, the touchscreen input must keep its size unchanged after encryption since the data is delivered to the mobile OS from QEMU through fixed-sized virtual registers. However, the size of register is too small to use AES block ciphers. Therefore, we need to use a different encryption/decryption algorithm. In our implementation, the virtual touchscreen leverages 2 4-byte registers to store the absolute values of the X and Y coordinates. As explained, the data after encryption should also be fitted into the 2 4-byte registers. To achieve this, we leverage the Prefix Cipher [38], 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 [38]. 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 algorithm because of its high efficiency. It requires only one table lookup to decrypt the message, which is necessary when the data volume is high.

Also note that one limitation of Prefix Cipher is that the size of the data cannot be too large, otherwise the mapping table becomes too big to accommodate. On the other hand, data size being too small weakens the encryption process due to lower brute force barrier. To balance the performance, we choose to divide the register data into 2-byte segments (with a table size of less than 1MB), and apply the Prefix Cipher algorithm on these segments. Recall that the virtual registers of touch screen read 2 4-byte input data in each poll, which can be exactly divided into 4 segments. Furthermore, since VTRUST allows dynamic key change through control data, we are able to install multiple keys to generate multiple mapping tables, and switch among them, to strengthen the security.

**Sensors** In mobile device, sensor data offers input from multiple dimensions and provides richer

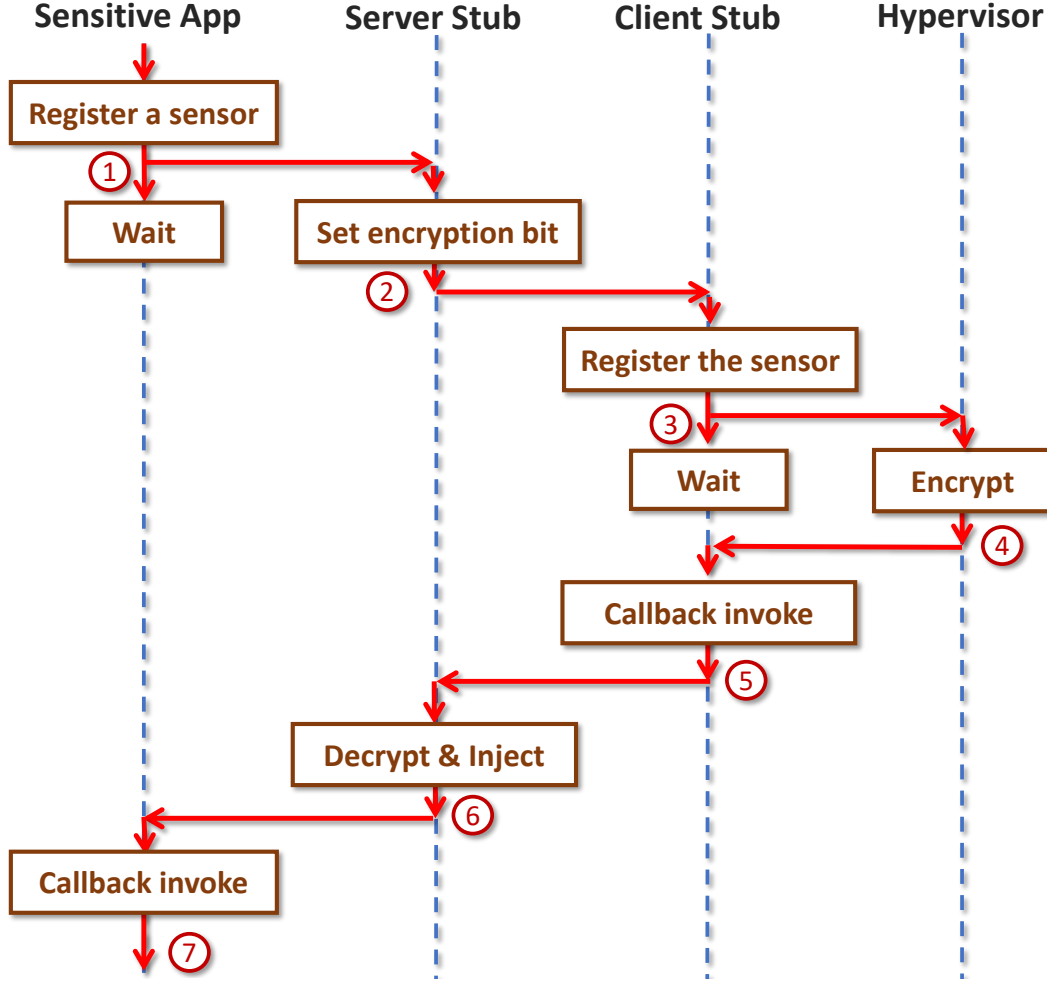
functionalities to mobile apps. Though the sensors bring a wealth of advantages, it has been shown that sensor data can be leveraged to launch various attacks (e.g., [45, 113, 118, 120, 153]). Therefore, vTRUST has to secure the sensor input.

The detailed steps of how vTRUST handles sensor input is illustrated in Figure 2.3. **Step ①** The sensitive app requests a type of sensor data, such as the accelerometer data. It needs to register a listener to the sensor manager in the Android system of the VM. **Step ②** Our modified sensor manager informs the Server Stub the request from the sensitive app. The Server Stub then sets the corresponding encryption bit of the control data. **Step ③** The control data is sent together with screen frames. Upon receiving the data, the Client Stub itself registers a listener of the same type of sensor to the local sensor manager. **Step ④** The hypervisor feeds the encrypted sensor data to the mobile OS. In this step, the encryption process is exactly the same as that of touchscreen data. **Step ⑤-Step ⑦** The data is relayed back to the sensitive app through Client Stub, Server Stub, and the sensor input system.

Unlike the touchscreen that has 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) [9] 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.

## 2.5 Evaluation

In this section, we present the evaluation result. Specifically, we evaluated the performance of vTRUST using an Arndale development board [128] as the mobile device since it adopts Exynos 5250 SoC, which supports hardware virtualization. Also, the Exynos 5250 SoC has a Samsung



**Figure 2.3:** Sensors Input Processing in vTRUST.

Exynos 5 dual core processor running at 2.0GHz with 2GB of RAM. Meanwhile, the development board runs a light-weight Linux as the host OS and establishes its virtualization environment using KVM and QEMU [144]. 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, 16GB RAM, and 3TB disk. In addition, the development board is equipped with an external accelerometer [6] and 7 inch touchscreen [14]. As for the server, we run Android-x86 VMs on the remote server by using the VMware workstation hypervisor.

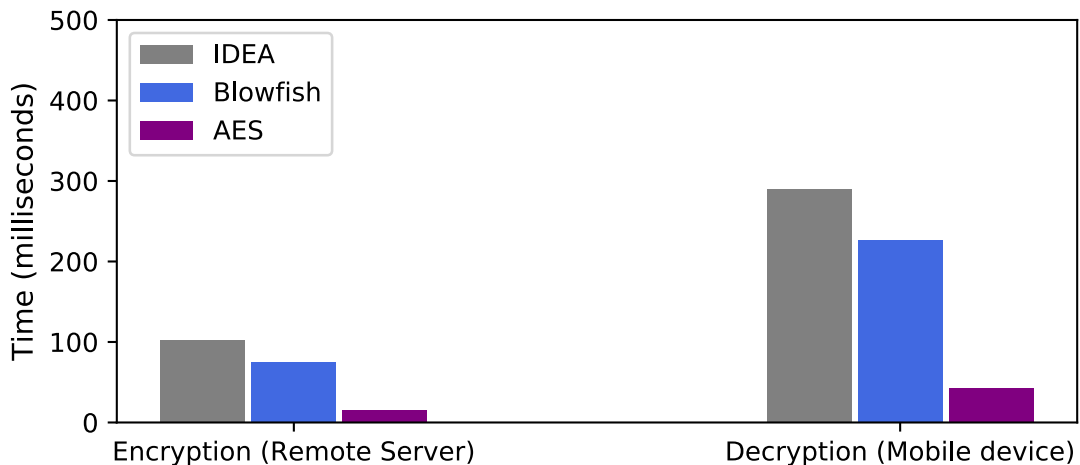
In our experiment, we first performed the microscopic measurement of vTRUST that includes the overhead from the encryption and decryption (§2.5.1), the compression and decompression (§2.5.2), and then at the macroscopic in terms of throughput (§2.5.3) and the responsiveness (§2.5.4).

### 2.5.1 Performance of Encryption/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 register 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 three encryption algorithms, which are IDEA, Blowfish and AES-128. 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 2.4.

Our experiments show that the five algorithms have significant differences regarding to computation overhead, which is stemmed from the algorithm complexity. We choose to use AES-128 eventually because of its excellent performance and perceived security. Another observation is that the encryption is faster than the decryption for each of the five algorithms, due to the fact that the server where the encryption is conducted is more powerful than the mobile device where the decryption is done.



**Figure 2.4:** Comparison of different encryption algorithms.

### 2.5.2 Performance of Compression/Decompression

In this experiment, we measured the compression ratios of our compression algorithm for different apps. Note that our compression mainly benefits from the unchanged segments of two consecutive frames. Therefore, our compression efficiency is significantly affected by the service nature of the app. For example, a video app, such as Youtube, may have lower compression ratio since the frames change more frequently.

To cover more use cases, we deliberately select 7 typical mobile apps from different genres, such as Chrome and Youtube, in our evaluation. We leverage the MobiPlay tool [124] 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. Figure 2.5 illustrates the box-plots of the compression ratios for the 7 apps.

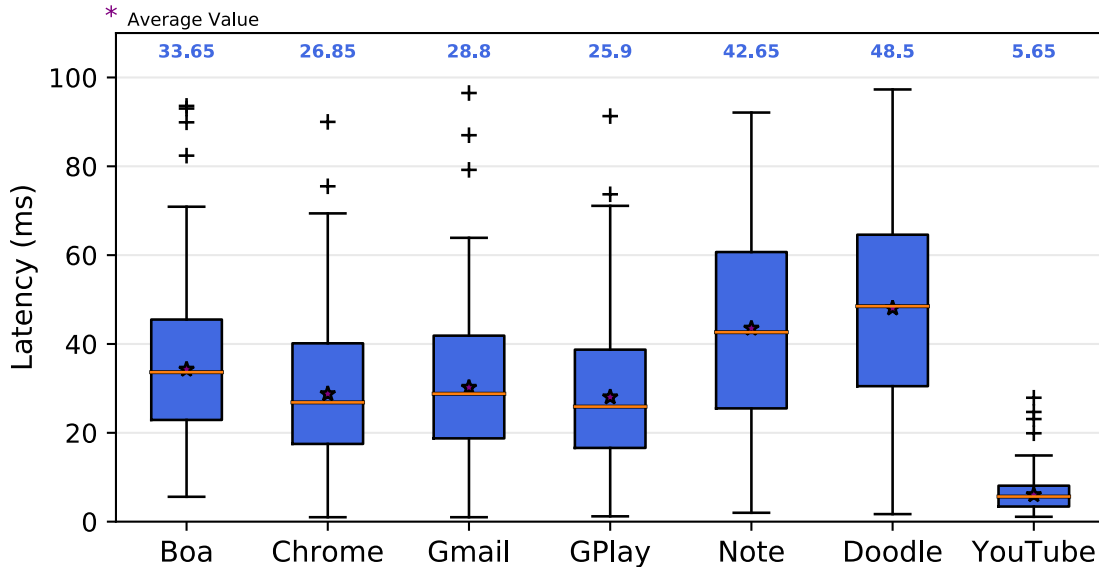
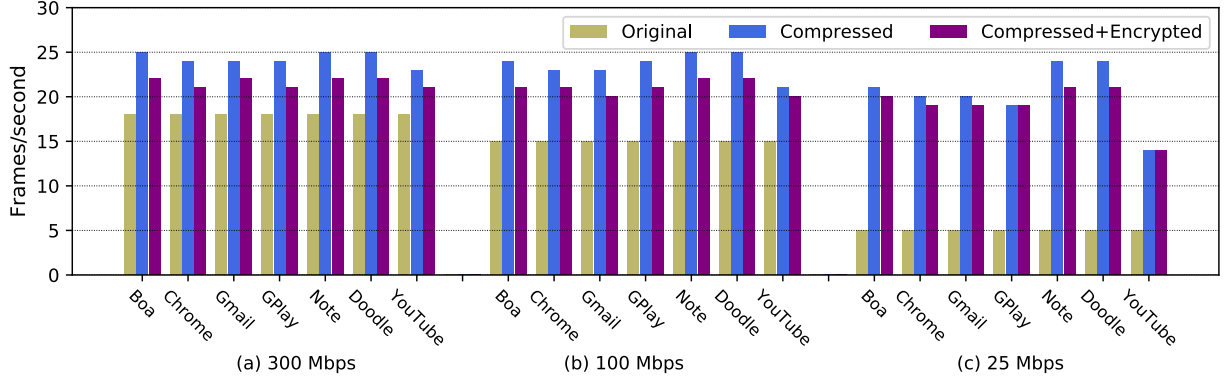


Figure 2.5: Compression ratios.

As shown in the figure, apps like Notes and Kids Doodle, whose screen content changes slowly, can achieve pretty high compression ratios, more than 42 : 1 on average. Mildly-changed apps (Bank of America Boa, Chrome, Gmail, and Google Play) can still get decent compression ratios, between 25 and 33 on average. In contrast, this number for video-centric apps (Youtube in our



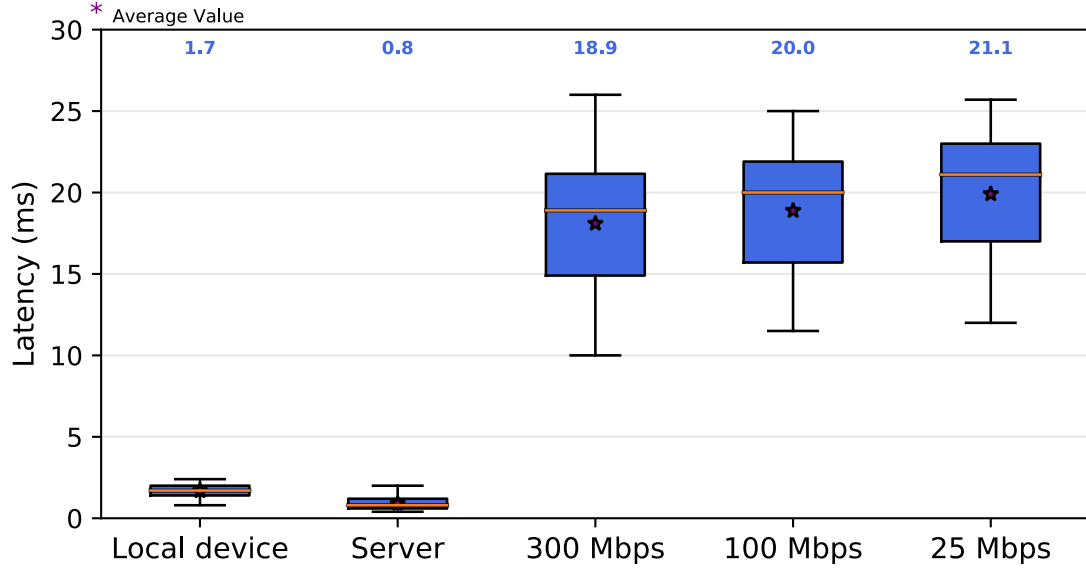
**Figure 2.6:** FPS of vTRUST output.

evaluation) is around 4 to 6. Though sometimes the compression ratio is limited, we show that it is still enough to produce acceptable frames per second and low latency as presented in §2.5.3.

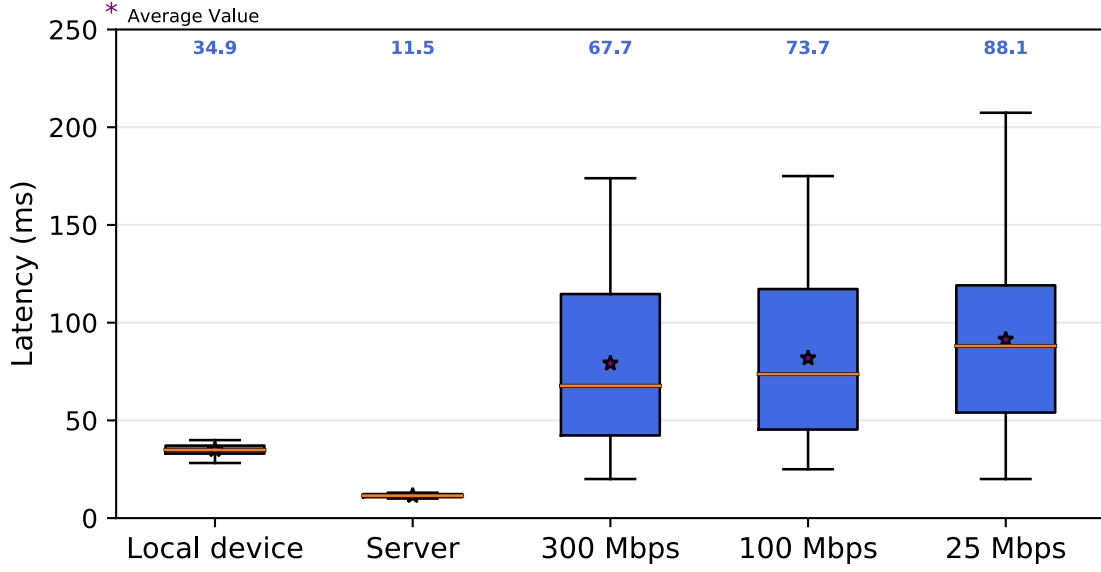
### 2.5.3 Throughput

In this evaluation, we measure the frame throughput of vTRUST. The frame throughput is defined as the number of frames from remote server VM that are shown on the mobile device screen in a second. Frame throughput is very important since it has direct impact on the user experience. A low frame throughput that results in jagged motions on the screen is considered a poor user experience. As the frame throughput is mainly affected by encryption overhead, compression ratio and network bandwidth, we tune our experimental settings against 3 different network conditions, in which bandwidth are configure to (a) 300 Mbps (b) 100Mbps, and (c) 25Mbps. For each network condition, we measure the throughput of frames in *original*, *compressed* and *compressed + encrypted* forms using the same apps and replay tools mentioned in §2.5.2.

Figure 2.6 shows that the throughput of uncompressed frame is significantly slower than compressed frame under condition (c) . With the help of our compression technique, vTRUST can maintain a relatively high throughput even under a low bandwidth condition. And the performance overhead introduced by encryption is acceptable – the largest frame drop we have observed is only 3FPS. Overall, we conclude that the frame rate of vTRUST is high enough to run non-video based apps. For video-based apps like YouTube, the frame throughput is relatively low (less than 15 FPS) under a low bandwidth condition. This is because our compression algorithm is less effective when



(a) Touchscreen latency



(b) Screen frame latency

**Figure 2.7:** vTRUST responsiveness.

the consecutive frames changes too much. More discussion on this will be provided in §2.7.

#### 2.5.4 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. To better understand the input latency, we break down the data flow and divide it into three stages:

- **Stage I** is from the hypervisor to the Client Stub on the mobile device. Specifically, the hypervisor reads the input data from the hardware and sends it to QEMU. QEMU encrypts the data and forwards it to the guest OS, and then to the client stub. The latency of this stage is reflected in the “Local device” box in Figure 2.7.
- **Stage II** is from the Client Stub to the Server Stub. This stage mainly involves network transmission. We measure the latency under 3 different settings as in our frame rate measurement. This part of latency is shown in the right 3 blue boxes in Figure 2.7.
- **Stage III** is from the server stub to the protected application on the remote server. Specifically, the server stub decrypts the received data and injects it into Android Services, which further delivers the data to the application. This part of latency is shown in the “Server” box in Figure 2.7.

To evaluate the input latency, we generate 500 touchscreen events (by pressing touchscreen randomly) on the client device under the 3 scenarios, and measure the time spent on each of the stages. Note that the sensor data is very similar to touchscreen data in size, and evaluating the touchscreen is representative for all input data. Figure 2.7(a) illustrates the results. We can see that the time consumed in stage 1 and stage 3 is negligible (both less than 3ms in average), and that the network latency in stage 2 is the dominant factor. However, as 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 hardly noticeable for users.

Similarly, we measure the output latency. The 3 stages in the opposite directions are considered and measured, of which the results are shown in Figure 2.7(b). In addition, the output is from a Youtube video, which has the largest output size after compression, and thus the largest latency. As expected, the latency in network transmission is higher. In addition, the time consumed by the mobile device is higher, which conforms to our observation in §2.5.1 that the server is much faster than the mobile device in encryption/decryption and compression/decompression. Even so, they still add



up to a small number, varying from 120ms to 140ms in average under different network conditions. This latency is noticeable. However, for most sensitive apps, it is still usable and fairly responsive. Furthermore, this experiment is done with the most resource hungry type of app (Youtube), one can expect much better performance with other more static apps. Thus, we conclude that vTRUST introduces a relatively small latency on remote apps.

## 2.6 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 §2.2.3, an attacker is able to access or modify all resources that the local mobile OS is entitled to. She is also able to eavesdrop or manipulate the network connections between the mobile device and the server. But, the attacker cannot read the memory of sensitive apps, or the storage of the VM, as the apps are running on the server, which is out of the attacker’s control. Therefore, sensitive data in both the storage and the memory of the VM is kept secure. However, there are still other attacks. In vTRUST, while the I/O between the mobile device and the server is encrypted, an attacker can try to probe or manipulate the I/O going through the mobile OS in order to mine sensitive information. Strategically, the attacker may either passively observe the data traffic or actively manipulate it.

More specifically, the attacker may try to probe and analyze the system by modifying the I/O data. Fortunately, in the case of screen frame output and touchscreen input data manipulation, the mobile user can immediately notice the attack, disconnect from the server, and report the incidence to the IT department. This is because the user can observe anomaly of the client stub behavior under these circumstances. For example, the user will find a distorted screen frame if the output data is modified. Or the user may find the touchscreen input does not fall on the correct positions. Sometimes, the sensor data does not generate obvious visual feedback to the user. For example, background apps like a pedometer can generate plausible results even if the data is altered. However, with slightly longer observation, it is still very likely to be noticed by the user, as these background

apps cannot always work as expected when consuming wrong data.

Finally, note that our design does not rely on the security infrastructure of the network such as SSL/TLS. Therefore, tampering the data in transit is equivalent of tampering data on the mobile OS, which we have already discussed.

## 2.7 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 [111, 118].

Second, the encryption mechanism can sometimes break the functionality of some non-sensitive apps. More specifically, when in Shield Mode, sensor data for remote apps are encrypted. However, other local apps may need that sensor data as well, which is now in the encrypted form and is sent to local apps before decryption. For example, a pedometer may record intensive random movement from a user even if she stands still. Currently vTRUST cannot resolve such an issue.

Third, 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.

Finally, 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 [18].

**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 [12] and VP9 [16]. 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 [61], in our remote VMs.

## 2.8 Related Work

**Computation offloading approaches** Many projects [53, 56, 70, 89, 110, 145, 152, 162] seek to protect the sensitive data with the assistance of a remote cloud. CleanOS [145] 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 [152] 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 [145]	✗	✗	✓	✗	✓	✓	✓	✗
TinMan [152]	✓	✗	✓	✗	✓	✓	✓	✗
TrustZone [130]	✓	✓	✓	✗	✓	✗	✓	✗
Overshadow [48]	✓	✓	✓	✗	✗	✓	✗	✓
OSP [52]	✓	✓	✓	✗	✓	✗	✗	✗
SGX [22]	✓	✓	✓	✓	✗	✗	✓	✓
VNC [15]	✓	✗	✓	✓	✗	✓	✓	✓
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 2.2:** 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 [35, 72, 129, 130, 142] 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 [49], especially when more SCC is put into the secure world [114].

**Hypervisor-based solutions** Some works [43] attempt to enforce security policies and provide TEEs with the aid of a hypervisor. Systems like Overshadow [48], CHAOS [47], SP3 [154] and InkTag [79] 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 [51]. While OSP [52] 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 [22] 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 [37, 42, 46, 138, 148] (e.g., SGXBOUNDS [92] for shielded execution and VC3 [136] 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 [15], SVMPP [8] and Rio [33], 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 2.2. We notice that vTRUST holds all of the capabilities compared, and the most closely related system is the VNC [15], especially from the user experience perspective. However, with VNC, untrusted operating systems are still able to view the I/O of the sensitive apps.

## 2.9 Chapter Summary

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 both 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 have implemented a prototype of vTRUST and conducted extensive evaluations. Our experimental results show that vTRUST introduces little impact on both user experience and the performance of mobile apps, especially for security sensitive ones.

## Chapter 3

# MobiPlay: A Remote Execution Based Record-and-Replay Tool for Mobile Applications

### 3.1 Introduction

In recent years, researchers have designed and developed several replay tools for mobile apps. However, none of them are able to truly capture all possible input. These tools can be divided into several categories. The first category is tools that obtain the input data by reading `/dev/input/event*` files through the Android SDK *getevent* tool (e.g., RERAN [67] and Mosaic [75]). Although these tools can record continuous gestures on the touchscreen (swipe, move, pinch/zoom), they come with several drawbacks. First, they entirely depend on whether the mobile phone’s OS provides interfaces to `/dev/input/event*`, which is not always the case. For instance, the Nexus 7 does not push any sensor data into any `/dev/input/event*` file. Second, they are unable to record sensors whose events are made available to applications through system services rather than low-level event interfaces, such as GPS. Third, they can only obtain the event data in low-level hexadecimal codes (e.g., 40-719451: `/dev/input/event4: 0003 0035 0000011f`), which is not human readable, hindering developers from white-box testing. Fourth, they have potential conflicts with other events occurring during a replay session. Another category is GUI-level tools, such as [29, 73, 125]. They work at a higher level of ab-

straction by capturing GUI objects, and usually require app modification (e.g., *android:debuggable = true*). Though they work well for discrete point-and-click GUIs, they cannot handle continuous touchscreen gestures or customized GUI elements.

A straightforward question is, "Can we solve all the problems and challenges within the mobile phone alone without modifying the operating system?" Unfortunately, the answer appears to be no. For security concerns, mobile operating systems, such as Android, *sandbox* apps in order to provide applications with the guarantee of isolation from other applications on the system. Each application has its own UID that prevents it from doing many things to other applications on the system. If we are to record all the input data for an app, we would have to develop a second app, without using any existing tools, such as the Android SDK *getevent* tool. However, as the Android Application Sandbox has enforced, the second recording app cannot access any data or memory of the app to be recorded. Therefore, we have to introduce other component, rather than the mobile phone alone, to solve this problem.

In this dissertation, we design a system, called MobiPlay, to record and replay an Android app's execution by introducing a server. The to-be-tested app is actually running on the server, while its GUI is transmitted back to the mobile phone as if the application were running locally on the phone. Although it may seem that latency would be a large concern in this setup, we find that the latency is acceptable due to the high-speed peer connection and the proximity of the server (we evaluate on a LAN connection). We have a video of MobiPlay on YouTube (search MobiPlay). Without modifying the mobile phone's operating system, MobiPlay is able to record all sensor data inputs, for replay later, in the form of high-level events, such as touchscreen gesture, key event, and sensor event. Besides solving the existing problems and challenges we outlined previously, MobiPlay is also able to offer more flexibility than ever before. It can not only record all input data, on both the mobile phone and the server side but it can also replay the app on both sides as well. Furthermore, our system is suitable for both white-box testing and black-box testing.

In summary, in this dissertation we make the following main contributions:

- We are the first to record input data of mobile apps in the application layer without modifying mobile phone's operating system, which is not achievable with the previous state-of-the-art

approaches.

- We have designed and implemented MobiPlay, a system that is able to record and replay the execution of mobile apps. Our system is richer than ever before because it is able to record all sensor data input.
- MobiPlay is able to simulate the same environment on the mobile phone and the server, which fundamentally expands the space of flexibility.
- MobiPlay is flexible in that it can record and replay on both the client (mobile phone) and server side.
- Our system enables white-box testing for app developers because it exposes high-level semantic events, and presents them in a human readable form instead of an encoded stream of raw hexadecimal event data.

The rest of the chapter is organized as follows. We present the whole system in Section 3.2 and describe the implementation in Section 4.5. Section 4.6 details the evaluation of MobiPlay. Section 3.5 briefly discusses the limitations and future work. We review the related work in Section 3.6 and conclude this chapter in Section 3.7.

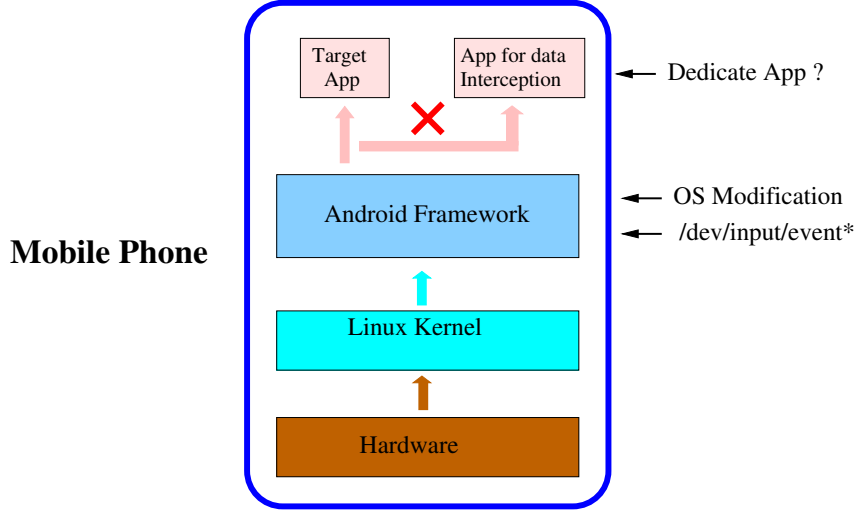
## 3.2 Design of MobiPlay

In this section, we will first elaborate on the rationale behind our design decisions. Then, we will explain our system design, its general architecture, and the details therein. Finally, we will explain the input data recording and replaying in our system.

### 3.2.1 Design Rationale

As mentioned in Section 4.1, we have to coordinate work between the user’s mobile phone and an external server to solve all of our design challenges. Here we further illustrate why the mobile phone **alone** cannot solve this problem, and why a server is indispensable. Figure 3.1 shows the logical flow of application input data. Suppose the user makes a gesture on the touchscreen (tap, swipe,





**Figure 3.1:** App input data flow, within a mobile phone (no server).

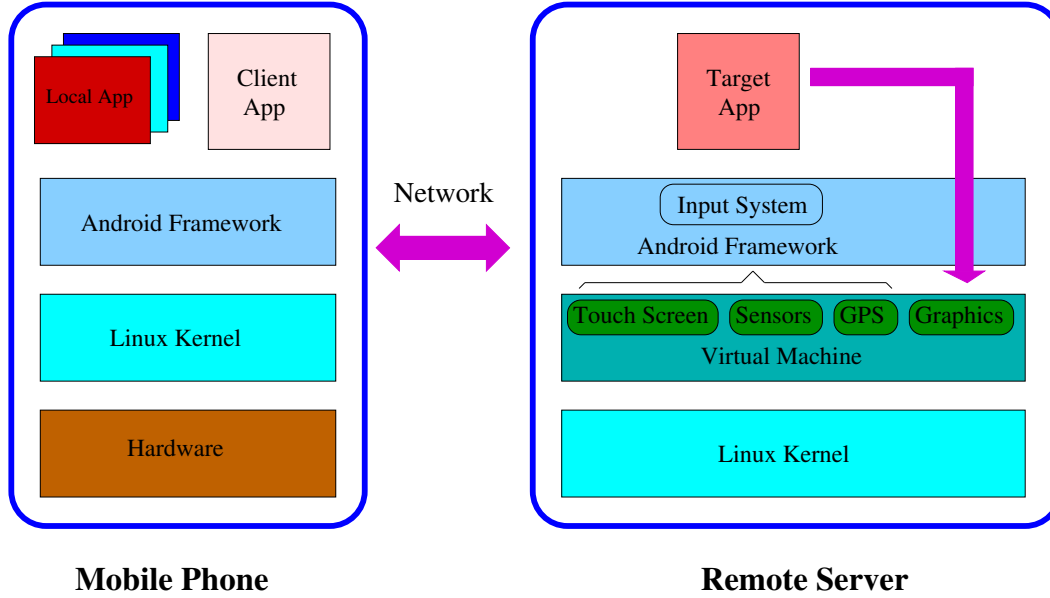
pinch, zoom, etc.). First, the touchscreen hardware captures this gesture, converts it into digital data, and informs the Linux kernel by sending an interrupt to the CPU. Second, after receiving the interrupt, the OS stops the current job, reads the input data with the corresponding driver, and sends the data to the Android framework. Third, the Android framework packages the data into discrete events (MotionEvent) and sends them to the related service, such as Sensor Service, Input Method Service, or Location Service; at the same time, it sends the data to `/dev/input/event*` as well in the form of hex codes. Finally, the related service sends the discrete events to the application running on the foreground.

RERAN and Mosaic obtain the app input data by reading `/dev/input/event*` files in the Android framework, and OS modification occurs in the Android Framework as well. From Figure 3.1, we can see that the only other possible location to record app input data, without modifying the OS, is in the application layer. In order to do this, one must develop another application, specifically dedicated to intercepting the input data that is actually destined for the target application. However, this is forbidden by the Android Application Sandbox, as well as the sandbox policy in other mobile operating systems, which guarantee that unsanctioned data sharing between applications is not possible. As a result, the dedicated data intercepting app cannot access any data or memory of the target application. Therefore, it is clear that a mobile phone alone cannot solve the challenges and problems that the current approaches have encountered. To overcome this “isolation” obstacle, we

introduce a second component, i.e., a sever.

### 3.2.2 Architecture of MobiPlay

In this chapter, we design MobiPlay, a client-server system consisting of a mobile phone and a server, as shown in Figure 3.2. The server and the mobile phone are connected through a high-speed network connection, like 300Mbps to 1Gbps.



**Figure 3.2:** MobiPlay consists of a mobile phone and a server.

In MobiPlay, there are two components associated with the application to be recorded and replayed (the target app, hereafter): a client app on the mobile phone and a virtual machine (VM) on the server. The target app runs on the VM on the server. The client app (the client, hereafter) is a typical Android app that does not require root privilege and is dedicated to intercepting all the input data for the target app. The VM is a “duplicated” mobile phone on the server, which has the same configuration as the physical mobile phone, including screen size, resolution, and all present sensors. The VM runs a modified Android operating system designed for x86 architecture. It is important to note that the tester/user has complete control over the server, including the modified Android operating system; specifically, she has root privileges, has access to modify and recompile the OS source code, is able to make configuration changes, etc.

The basic idea of MobiPlay is that the target app actually runs on the server, while the user interacts with the client app on the mobile phone. The user is not explicitly aware that she is, in effect, using a thin client. At the beginning, we install the target app on the virtual machine on the server, and the client on the mobile phone. The client shows the GUI of the target app in real time on the mobile phone, exactly as if the target app were running on the mobile phone. As a result, the user just needs to interact with the target app as usual, while, under the surface, the client continuously forwards all input data (such as touchscreen gestures, sensor data, and GPS) to the VM on the server. At the same time, the GUI of the target app on the server is forwarded to the client and is then displayed by the client on the mobile phone. The VM on the server injects the input data received to the related OS services, which in turn send it to the target app. The target app runs on the virtual machine with the injected input data exactly same as it would run on the mobile phone. In other words, the user runs the app with exactly the same experience as if the app had been running on the mobile phone. The target app actually runs on the virtual machine on the server, but, with the same environment (inputs, resolution, screen size, etc.) as if it had been running on the mobile phone.

MobiPlay has three modes: *normal*, *record*, *replay*. When the client is opened, the graphical interface presents three buttons to the user, and the user must choose one of the three modes before establishing a connection with the server. In the normal mode, MobiPlay runs without any data recording or data replaying. That is, the user of the mobile phone runs the mobile app on the server, while interacting with the local client app. The mobile user can use this mode to test MobiPlay. Another possible application scenario for this mode might be that the mobile user wants to offload a resource-hungry app to the much more powerful server. In record mode, MobiPlay intercepts all input data through the client app, as the target app runs on the server. The collected data is stored on disk. A user configuration option allows for the data to be stored on the phone, or the server, or both. In the replay mode, MobiPlay first configures where to read data (phone or server), where to replay (phone or server), and the test type (black-box or white-box). Then it reads the input data from disk, injects the input events, in the same order, into the target app running on the chosen device, and replay the target app. During the replay, MobiPlay does not process any local Android

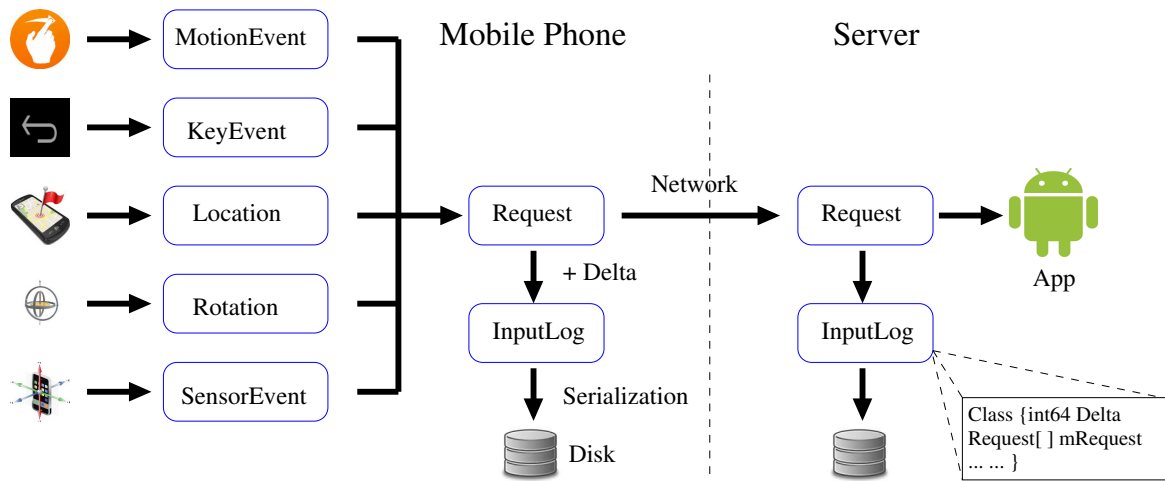
service request, such that the target app is replayed with the recorded input only. This avoids any interference from the current state (such as new GPS or accelerometer data).

In the following, we will describe the data recording and the app replay mechanisms in more detail.

### 3.2.3 App Recording

When the user chooses record mode, MobiPlay will be directed to record all input data for replay later. MobiPlay can record all input data for the target app on both the mobile phone and the server. As illustrated in Figure 3.2, all input data for the target app passes through the client app on the mobile phone. Therefore, the client is able to intercept all this data. At the same time, since all the input data is transmitted to the server and the user has full control of the server, the data can be intercepted and recorded there as well.

Figure 3.3 shows how the data is intercepted and stored on both sides. On the phone side, all



**Figure 3.3:** MobiPlay records input data on both the mobile phone and the server.

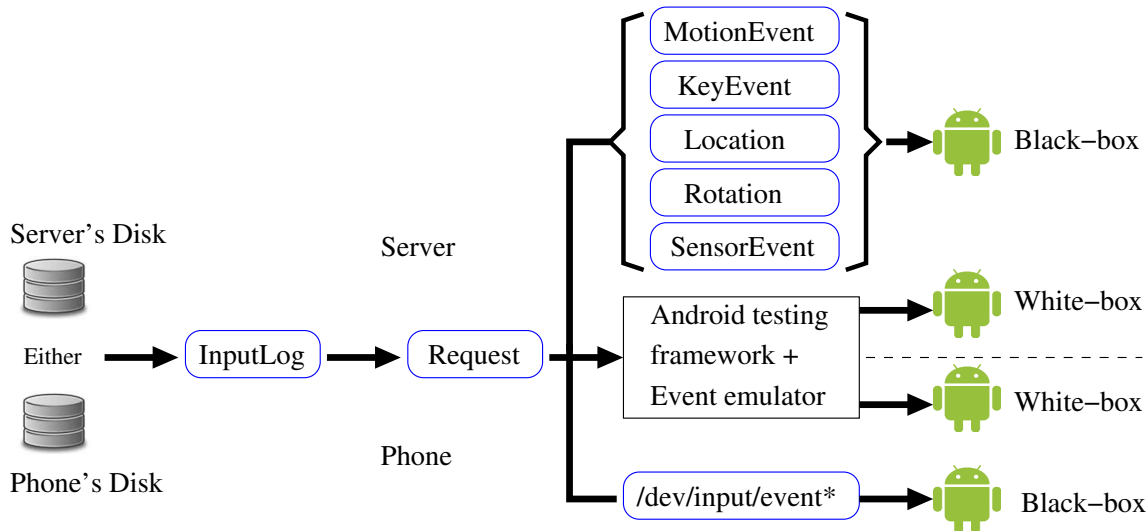
the data is intercepted in the form of events, such as motion events, key events, location, rotation, sensor, etc., and each event is an object that contains the input information at a certain point of time. MobiPlay extracts useful information from each event and stores it in a structure called *Request*. MobiPlay also obtains the time intervals between each pair of consecutive events, and stores this information, together with the Requests in order, in the log. The log itself is a class

called an *InputLog*, which is a collection of Request objects. Finally the entire InputLog instance is written to disk on the mobile phone. At the same time, all Requests are transmitted through the network to the server, in order to be fed into the target app. Thus, MobiPlay is able to record all input data on the server side as well. Once the target has finished running, i.e, the recording procedure has completed, MobiPlay quits to the GUI with three modes for selection.

One prominent advantage of this architecture is that our recorded data is high-level, readable and revisable. This is incredibly valuable for app developers when trying to recreate and fix bugs in their code caused by specific, infrequent input.

### 3.2.4 App Replaying

When the user chooses replay mode, MobiPlay will load the recorded data from the disk and replay the target app. The replay procedure is illustrated in Figure 3.4. The replay procedure for the white-testing on the mobile phone is similar to that on the server, and neither of them needs root privilege. However, the replay procedure for the black-box testing on the mobile phone is different from that on the server. In case on the phone, the black-box testing still requires root privilege, since the input data has to be injected through `/dev/input/event*` files.



**Figure 3.4:** MobiPlay can relay an app on both mobile phone and server, black-box and white-box testing, respectively.

As Figure 3.4 shows, first, MobiPlay loads the input data from the disk of either the server or

the mobile phone. Second, the input data stored on disk as an InputLog class is unpacked into a sequence of Request (i.e., events). Third, (a) for black-box testing on the server, the sequence of events is sent to virtual devices on the VM, which then inject events to the target app for replay; (b) for white-box testing on the server, the input events is injected to the target app through Android testing framework and event emulator; (c) for white-box testing on the phone, it is same as the white-box testing on the server; (d) for black-box testing on the phone, the events is converted into hex codes and fed into the corresponding `/dev/input/event*` files, where the target app read the input data for replay. After the replay has finished, MobiPlay quits to the GUI with three modes for selection.

MobiPlay is advantageous in that app developers can revise the input data as they want to test the app in different scenarios. This is much easier than re-running the app multiple times to collect input data, hoping for good test coverage from all types of input.

### 3.3 Implementation

In this section, we present our implementation of MobiPlay. We establish the client-server platform by leveraging the Secure Virtual Mobile Platform (SVMP) [8], and then we build the record and replay approaches on basis of SVMP.

#### 3.3.1 Physical Devices

In principle, MobiPlay only consists of a mobile phone (or a tablet) and a server by utilizing the existing networking infrastructure. In our implementation, besides the phone and the server, a router is used to set up the wireless connection between the server and the phone. The characteristics of all these devices are listed in Table 3.1.

Additionally, the server uses VirtualBox<sup>1</sup> as the VM hypervisor and uses a virtual bridged network adaptor for networking access. The VM configuration allocates 4096 MB RAM, an 8 core processor, and 5GB disk storage space. And the VM runs Android OS v4.4.4.

---

<sup>1</sup><https://www.virtualbox.org/wiki/VirtualBox>

**Table 3.1:** The devices used in MobiPlay system.

Device	Specification
<b>Android phone</b> Samsung Galaxy S4	quad-core 1.6GHz Cortex-A15 quad-core 1.2GHz Cortex-A7 2GB RAM, 32 GB microSD Android OS, v5.0.1
<b>Android tablet</b> Nexus 7, 2013	quad-core 1.5GHz Krait 2GB RAM, 32GB storage Android OS, v4.1.1
<b>Server</b> Y480 Lenovo laptop	2.4GHz Intel i7-3610QM 8GB RAM, 500 GB HD Ubuntu 14.04
<b>Router</b> TP-Link TL-WR841N	300Mbps

### 3.3.2 The Client-Server Platform

In MobiPlay, the essential component is the client-server platform, which we implement using SVMP. SVMP is a secure mobile application platform developed by MITRE<sup>2</sup>, based on thin client technology and cloud computing technology. An open source “virtual smartphone”, SVMP runs an Android-based mobile operating system on a cloud platform.

In the big picture, we utilize SVMP to create a virtual machine on the server, where the target mobile app actually runs, and an SVMP client on the mobile phone, where the GUI of the app is displayed in real time. The client and the virtual machine are connected through a wireless network in our setup, which is common for mobile devices.

#### 3.3.2.1 SVMP Client

The SVMP client is installed on the mobile phone as a normal mobile application. The client, simple and unprivileged, is associated with the VM on the server rather than the target app. That is, when we test multiple apps, we just need to install each app in turn on the VM, without making any changes to the client. While MobiPlay runs in *normal* or *record* mode, the client captures native touch screen events, sensor inputs like the accelerometer and gyroscope, location information, and messages such as notification pop-ups and Android “Intents”. All these data is packaged under the

---

<sup>2</sup>[www.mitre.org](http://www.mitre.org).

SVMP message protocol, and is sent from the client to the server in real time. At the same time, the client displays the GUI of the target app transmitted from the server. In a word, the client enables the user to interact with the target mobile app running on the server, in the same way as if the app had been running on the mobile phone.

### 3.3.2.2 SVMP Virtual Machine

The SVMP VM is installed on the server. On top of the VM is the Android framework where the target app is installed. The VM provides virtual devices including *Touch Screen*, *Sensors*, *GPS*, and *Graphics*, which can be seen in Figure 3.2. The first three virtual devices are responsible for feeding the input data, as captured on the client side, to the target app running on the VM. As the target app runs, its GUI is displayed on the virtual display, i.e., *Graphics*. The VM, in turn, packages whatever is displayed on *Graphics* and sends it to the client in real time.

It is important for MobiPlay to maintain the same environment on the VM as that on the mobile phone, including the screen size, the resolution, and all the input devices. For instance, if the screen size is different, the touchscreen gestures will be represented with coordinates that are incorrect, or even undefined, on one of the displays. In our implementation, we ensure that no device mismatch occurs in our system to avoid problems like these.

### 3.3.2.3 Networking

It is critical to maintain a high-speed network connection between the mobile phone and the server, otherwise the user experience of MobiPlay will be impacted. The network is responsible for transmitting the input data from the client on the phone to the VM on the server and the GUI of the target app from the VM to the client in real time. As the mobile phone does not have a wired-network option, we set up a wireless network using a TP-Link TL-WR841N router, which can provide connectivity with throughput up to 300Mbps. Fortunately, SVMP uses WebRTC<sup>3</sup> to transmit data between the phone and the server, which greatly reduces the latency and the data volume. This is especially useful for situations where there may only be a slower network connection option available. In our implementation, MobiPlay maintains a frame rate of 50 FPS.

---

<sup>3</sup><http://www.webrtc.org/>



### 3.3.3 The Record Approach

As mentioned in Section 3.2.3, MobiPlay is able to record the input data in order to replay it later, on both the mobile phone and the server. Here we detail the implementation of the recording procedure.

On the mobile phone side, the client in MobiPlay intercepts all input data for the target mobile app in the form of events, which are grouped into five categories: **MotionEvent**, **KeyEvent**, **Location**, **Rotation**, and **SensorEvent** (please refer to Figure 3.3). Each event is an object containing the input information at a certain time. After intercepting an event, the client extracts only the necessary information for replay, and creates a data structure called a *Request* to store it. The client also calculates the time interval between two consecutive events. The event information and the time interval are then logged in an *InputLog* class, which stores a collection of *Request* objects. Finally, the *InputLog* is serialized and stored on disk.

Note that we define the *InputLog* class via Google’s protocol buffer<sup>4</sup>, which is a language-neutral, platform neutral, extensible, and automated mechanism for serializing structured data. The data stored in *InputLog* can be easily converted to JSON or XML format which is human-readable.

In the following, we will describe the interception of each of the five categories of events.

#### 3.3.3.1 MotionEvent

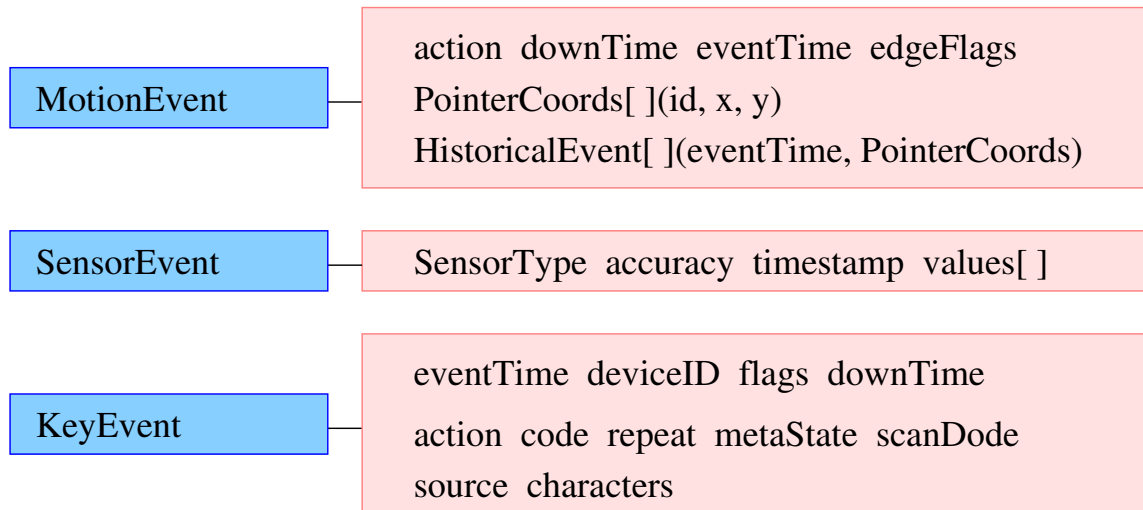
For most mobile apps, the most frequent input data is touchscreen events, including tap, press and hold, pinch, zoom, swipe, etc. The Android framework uses the *MotionEvent* class to record each touchscreen event. *MotionEvent*s describe movement in terms of an action code and a set of axis values. The action code specifies the state change, such as a pointer going down or up. The axis values describe the position and other movement properties.

*MotionEvent*s occur representing all possible touchscreen actions, such as *ACTION\_DOWN*, *ACTION\_MOVE*, *ACTION\_UP*, the time, the coordinates, and any historical event that happens before the current event. For the purpose of efficiency, Android may batch multiple touchscreen events into a single *MovementEvent* with several movement samples, and an *ACTION\_MOVE*

---

<sup>4</sup><https://developers.google.com/protocol-buffers/docs/overview>

action code. MotionEvent are passed as a parameter into the onTouchEvent() method, which is triggered by Android framework when a touch gesture happens. The top panel of Figure 3.5 shows all the fields of the class MotionEvent, which MobiPlay intercepts and records.



**Figure 3.5:** The MotionEvent, SensorEvent and KeyEvent classes along with their associated fields.

### 3.3.3.2 SensorEvent

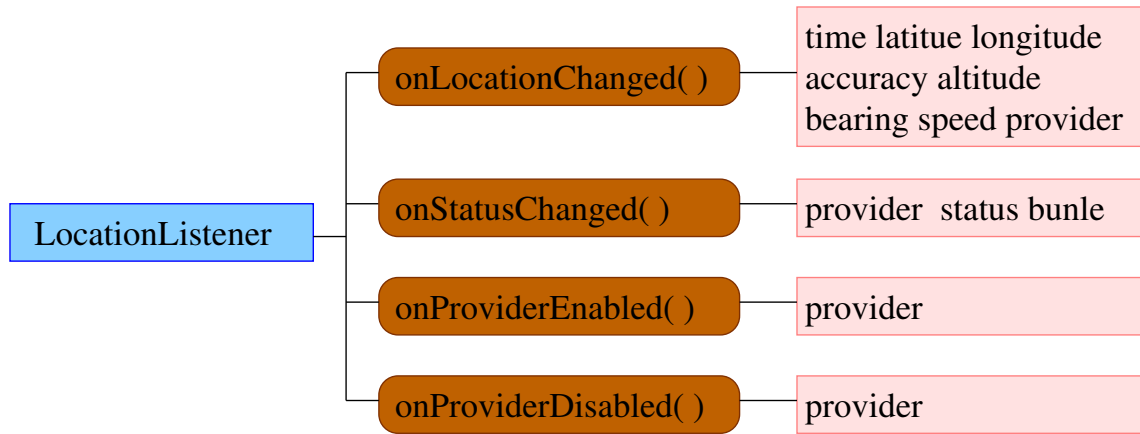
In the Android framework, sensor data is represented in the SensorEvent class and as each sample occurs, a SensorEvent instance is sent to the app by the sensor service. Each SensorEvent contains four fields: sensor type, time, accuracy, and the new data value(s), as shown in the middle panel of Figure 3.5. In our implementation MobiPlay can support the following physical sensors: accelerometer, gyroscope, light sensor, magnetic sensor, pressure sensor, proximity sensor, and virtual sensors (gravity, linear acceleration, orientation, and rotation vector). Sensor events are intercepted by onSensorChanged() method in the client app.

### 3.3.3.3 KeyEvent

KeyEvent is used to report key and button events, and is intercepted by the dispatchKeyEvent() method. Each key press is described by a sequence of key events. All information for these events is listed in the bottom panel of Figure 3.5.

### 3.3.3.4 Location

One advantage of our MobiPlay system is that it can handle location data (i.e., GPS), which other current approaches, such as RERAN and Mosaic, cannot. MobiPlay utilizes the `LocationListener` class to intercept location information. `LocationListener` has four methods, each of which has several input parameters, as listed in Figure 3.6. Whenever changes have been made to the location, Android framework will trigger the `LocationListener` to notify the target app. At the same time, it intercepts and records all these parameters.



**Figure 3.6:** The class `LocationListener` and its four methods with the corresponding parameters.

### 3.3.3.5 Rotation

MobiPlay can record device rotation changes as well. If the app runs on the mobile phone, whenever any rotation change is detected, the Android framework will notify the app and trigger the GUI to change from *portrait* to *landscape* or vice versa. The advantage of this design is that the app itself does not need to monitor the orientation of the gravity sensor, which eases work for developers who only need to maintain the state of their GUI when transitioning between view orientations. In MobiPlay, however, the target app will not obtain any rotation event because the target app is running on the server and the Android framework on the server will not detect any rotation change. To solve this problem, MobiPlay collects all rotation events by leveraging `onOrientationChanged()` in the client app on the mobile phone. Besides black-box testing, these events are also useful in white-box testing, because we can use these events to change the screen orientation. The rotation

event has only one parameter, *orientation*, ranging from 0 to 359 degrees. Specifically, 0 degrees means that the device is oriented in its natural position, 90 degrees means its left side is at the top, 180 upside down, and 270 indicates the right side is on top. When the device is nearly flat (parallel with the ground), the orientation cannot be determined, and `ORIENTATION_UNKNOWN` will be returned. For efficiency, MobiPlay does not record this undetermined case. It is worth pointing out that RERAN cannot handle rotation events.

### 3.3.4 The Replay Approach

MobiPlay is able to replay a mobile app on both the server and the phone, for both black-box testing and white-box testing. In the following, we will first present the procedure of the replay on both sides. Note that in the replay duration, MobiPlay does not process any new input from local Android services in order not to interfere the replay; though it still can receive them from local Android services, it does not send it to the app. We then will describe an event-sampling technique to revise the replay data as the tester wants.

#### 3.3.4.1 Replay on the Server

The replay procedure for black-box testing on the server is quite similar to what happens on the server in the *normal* mode except (1) the input is read from the log stored on disk, instead of from the client directly. Also, (2) all concurrent input from mobile phone is discarded. The input data can be injected from either the disk of the mobile phone or that of the server.

At the beginning, the user sets the mobile phone in replay mode. When the client detects that the mobile phone is currently in replay mode, it will load the corresponding input data from disk and store it in an `InputLog` class, which has the same form as the class used in data recording, by calling the `parseFrom()` method in the Google protocol buffers. The events are simply injected into the system via the virtual devices. To replay the app correctly, it is of the utmost importance to keep the events in the correct order. As mentioned in the data recording procedure, the recorded data includes all the information of every event and the inter-arrival time between all pairs of consecutive events. We need to adjust the time information in the recorded data according to the current time.

Specifically, for the first event, MobiPlay changes the event time to the current time, and adjusts the subsequent events accordingly. Then, MobiPlay packages the events into a Request, the same structure used in data recording. After sending the first Request to the target app, MobiPlay will wait in order to maintain the correct inter-arrival time between events. Then the next Request object is sent and the process is repeated. This continues until all input data has been read from disk and injected. After the replay has finished, MobiPlay will automatically switch back to the GUI with three modes for choosing. Algorithm 3 summarizes the whole procedure.

---

**ALGORITHM 3:** The replay procedure of black-box testing on the server.

---

```

/* InputLog Class {int64  $\delta t$                                 Request[] mRequest} */
Input: disk=phone/serve;
        device=server;
        test=black-box;
if disk==phone then
|   read input data from phone's disk;
else
|   read input data from server's disk;
end
store data into InputLog by call parseFrom();
while Request is not empty do
|   (event,  $\delta t$ )=getNextRequest(InputLog);
|   change the time of event to current time;
|   inject event to the app;
|   sleep( $\delta t$ );
end
Return to the GUI for mode selection;

```

---

Since white-box testing is more meaningful on the phone than on the server, and the procedure is similar on the server and on the phone, we defer white-box testing details to Section 3.3.4.2.

### 3.3.4.2 Replay on the mobile phone

For some tests, we may want to conduct the replay on the mobile phone itself. As we know, MobiPlay records all input data when the target app is actually running on the virtual machine on the server. Since the virtual machine offers the same environment as the mobile phone, including screen size and resolution, the recorded data can be replayed on the mobile phone. However, in this case, the tester does not have full control of the mobile phone. Therefore, the replay procedure is

different. In the following, we will describe the replay of black-box testing and that of white-box testing, respectively.

**Black-box testing.** For black-box testing, we cannot make any change to the target app. As analyzed in the introduction, the Android Application Sandbox forbids one application from injecting data into any other application. The only way to inject the input data for replay is to leverage the *sendevent* tool, which requires root privilege. Therefore, for replay on the mobile phone, MobiPlay runs into the same limitations as RERAN does. As the recorded input data in MobiPlay is high-level events, we need to convert it back to hex codes first, and then inject it by writing into the corresponding `/dev/input/event*` files.

**White-box testing.** Here, we want to inject the replay data directly into the target app assuming that we have the source code of the target app, with the goal of modifying the target app as less as possible. In a normal Android system, the Android framework communicates with the app through API, and sends the input data to the app in the form of events. Therefore, in white-box testing, the recorded input data also has to be sent to the app in the form of events. With the recorded data, we need to recreate all five events: `MotionEvent`, `KeyEvent`, `SensorEvent`, `Location`, and `Rotation`. Unfortunately, we cannot create a `SensorEvent` object as we do for the other four events because `SensorEvent()` is not public in *android.hardware.SensorEvent* class. Thus, we define a new class, `NewSensorEvent`, to carry sensor data (i.e., `SensorType`, `accuracy`, `timestamp`, and an array of values).

Depending on whether to modify the app, the five events are grouped into two categories. The first category, `MotionEvent` and `KeyEvent`, does not need to modify the app. Android provides its own testing framework called “the Android testing framework”, which is well integrated into the Android SDK tools. It offers powerful and easy-to-use tools that help developers test their applications at every level, from unit to framework. We use the instrumentation class in this testing framework to inject `MotionEvent` and `KeyEvent` through `SendPointerSync()` and `SendKeySync()`, respectively. The second category, `SensorEvent`, `Location`, and `Rotation`, requires to modify the app since the Android testing framework does not provide corresponding APIs. We have to manually inject these events into the target app. Specifically, for `Location`, we call four methods under

the Android testing framework: `onLocationChanged()`, `onStatusChanged()`, `onProviderEnabled()`, and `onProviderDisable()`; for Rotation, we call `onRotationChanged()`; for SensorEvent, we overload `onSensorChanged()` method and then call it. Table 3.2 summarizes how these five types of events are injected.

**Table 3.2:** Details of data injection in white-box testing.

Type	Recreated?	Modify app?	Injection description	Injection method
MotionEvent	Yes	No	Use instrumentation class in Android testing framework.	SendPointerSync( )
KeyEvent	Yes	No		SendKeySync( )
SensorEvent	No	Yes	Developers need manually inject these data to the target application.	Call onSensorChanged( )
Location	Yes	Yes		Call four functions
Rotation	Yes	Yes		Call onRotationChanged ( )

### 3.3.4.3 Event Sampling

The input data for replay is a sequence of events, each of which has a timestamp. We can consider these events as samples, and re-sample them at times different from those at which they are originally captured. In the InputLog, we record the events and the time interval of each pair of consecutive events. What we need to do is to change the time intervals and the event time accordingly without affecting the correct execution of the app.

First, we can use event sampling to cancel the latency introduced by the server. There is inevitably a latency from the point of view of the mobile phone, since the app actually runs on the server, even though the latency is small. To cancel the latency, we can shrink the time interval between every pair of consecutive events, say event *a* followed by event *b*, by the amount of event *b*'s latency; please refer to Section 3.4.2 where the latency of different types of input has been measured. To do this, MobiPlay carefully examines the events and identifies which are affected by the latency and which are not, and only adjusts the time intervals associated the former. Note that MobiPlay adjusts the event time of each event as well according to the shrunk time intervals.

Second, we can replay an app in fast mode with MobiPlay by adjusting the time interval between two consecutive activities, similar to the technique used in RERAN. For instance, imagine that the user zooms in and then clicks a button on the screen, we can shorten the time interval between the two activities when replaying the app.

## 3.4 Evaluation

In this section, we will evaluate MobiPlay. First, we demonstrate that MobiPlay can record and replay a variety of mobile apps. Then, we measure the latency introduced by the server, and the time/space overhead. Finally, we will test the event sampling technique.

### 3.4.1 Usability

For usability, MobiPlay currently does not support mobile apps that require ARM-based third party libraries, since the server in MobiPlay is x86-based. MobiPlay does not support camera or 3D acceleration either. As listed in Table 3.3, we randomly tried 52 apps from Google Play in different categories, including games, tools, news, health & fitness, lifestyle, education, shopping, etc., none of which requires ARM-based third party libraries or 3D acceleration libraries. We have successfully recorded and replayed all of the 52 apps (the replay is done on the server side multiple times).

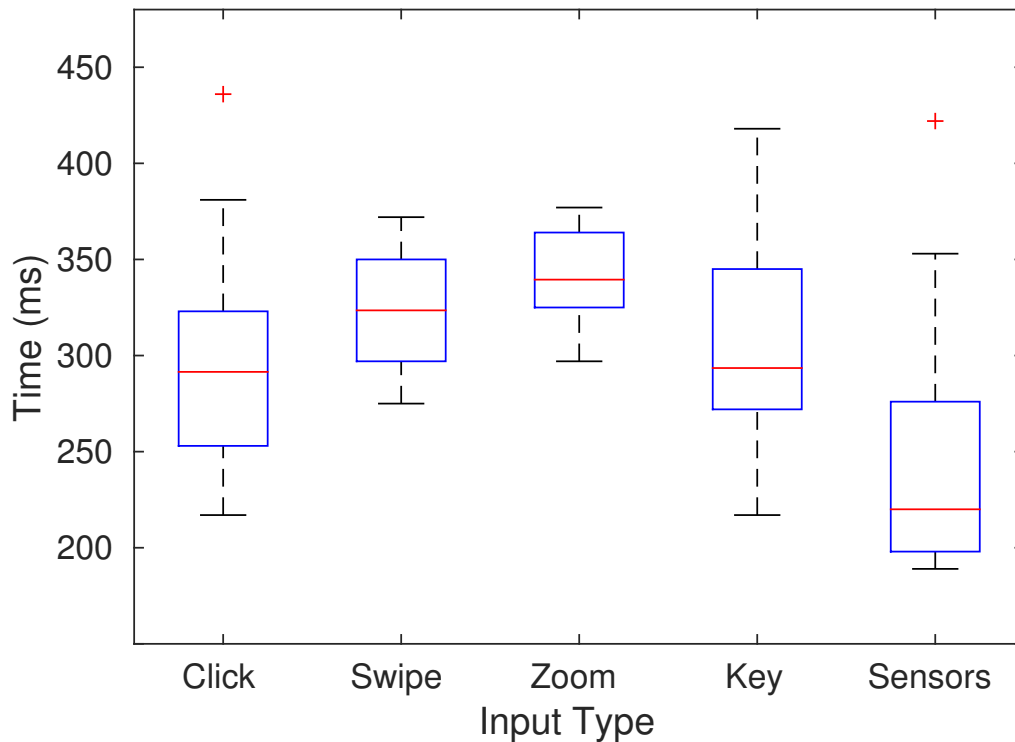
**Table 3.3:** The apps that MobiPlay has recorded and replayed successfully.

Name	Category	Name	Category
Exploration Lite	Adventure & Creativity	Cartwheel by Target	Lifestyle
Bible	Book & Reference	Instructables	Lifestyle
Amazon Kindle	Book & Reference	MyChart	Medical
Bing Search	Book & Reference	NBC news	News & Magazines
Concur	Business	BBC News	News & Magazines
Square Register	Business	CNN News	News & Magazines
Kids Doodle	Casual	Reddit is fun	News & Magazines
ZingBox Manga	Comics	Flipboard: News Magazine	News & Magazines
Crunchyroll Manga	Comics	Hola Launcher	Personalization
TeachersPayTeachers	Education	Iron Man 3 Live Wallpaper	Personalization
Math Expert	Education	photo editor	Photography
Bing Dictionary (ENG - CHN)	Education	Emoji Keyboard	Productivity
Chase Mobile	Finance	Evernote	Productivity
Mint: Personal Finance & Money	Finance	Onet Connect Fruit	Puzzle
Bank of America	Finance	Amazon for Tablets	Shopping
Tic Tac Toe Free	Game	Best Buy	Shopping
Bubble Shooter Classic	Game	Meetup	Social
Crush Eggs	Game	NFL Fantasy Football	Sports
Word search	Game	Sensor Box for Android	Tools
Chinese Checkers Wizard	Game	Sensors	Tools
Pedometer	Health & Fitness	File Manager	Tools
Calorie Counter - MyFitnessPal	Health & Fitness	Shell Terminal Emulator	Tools
Noom Walk Pedometer	Health & Fitness	Clock	Tools
Cardboard	Libraries & Demo	Adobe AIR	Tools
Always Positive -Daily Quotes	Lifestyle	Amber Weather	Weather
DIY Garden Ideas	Lifestyle	The Weather Channel	Weather



### 3.4.2 Latency

The client-server model introduces latency. From the view of the mobile user, she cares about how long it takes to get a response for her input. For instance, suppose she clicks a button on the client, then how long does it take till she notices that the click really happens? Thus, we define the latency as the time interval between the time the input occurs at the mobile phone and the time the input takes effect on the mobile phone; that is, the round-trip time of the input between the mobile phone and the server. To measure the latency, we have designed an app such that the screen turns red when an input event finishes. Then we run the app on MobiPlay and record the time when an event occurs on the mobile phone and the time when the screen turns red. For instance, for the event of *click*, we record the time when ACTION\_UP of click occurs and the time when the click spot turns red.



**Figure 3.7:** Round-trip time for different types of input.

We have evaluated the latency for five different types of input, each with 10 rounds. Figure 3.7

show the box plot of the results. As we can see, all the latency is less than 450 milliseconds, and the average is below 350 milliseconds, which does not affect the continuity of app execution and is acceptable to most testers. Individually, the *sensor* input has the shortest latency; the reason is that MobiPlay only needs to intercept four fields of data, as listed in Figure 3.5. The inputs of *swipe* and *zoom* have longer latency; one reason is that MobiPlay has to intercept more data including historical data (refer to Figure 3.5) and SVMP also batches a sequence of actions. The input of *click* and that of *key* have nearly the same latency, shorter than that of *swipe* and *zoom* but longer than that of *sensor*. Even though they belong to different categories of events (MotionEvent and KeyEvent, respectively), they share the same action with an ACTION\_DOWN and an ACTION\_UP, leading to similar latency.

### 3.4.3 Time and Space Overhead

**Table 3.4:** The time and space overhead and number of events in each category.

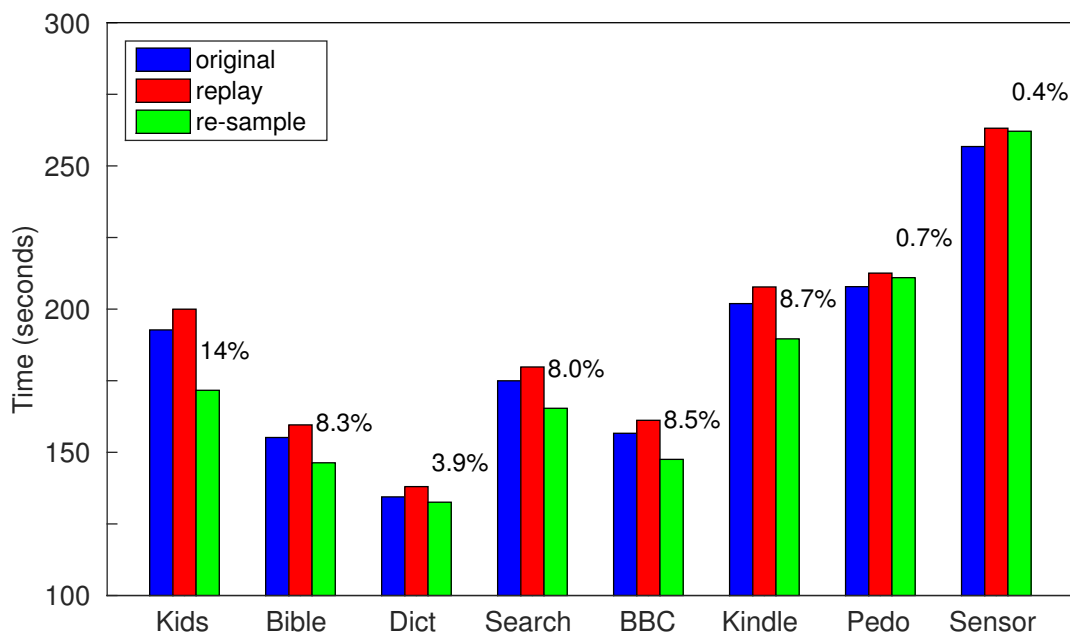
App name	Running time (seconds)			Data size (KB)	# of ME	# of KE	# of SE	# of R	# of L
	Original	Replay	Overhead						
KidsDoodle	192.74	199.98	3.7%	381.5	4571	4	0	0	0
Bible	155.17	159.58	2.8%	171.6	1989	8	0	0	5
Bing Dictionary	134.45	138.03	2.7%	78.9	909	20	0	0	0
Bing Search	174.98	179.80	2.8%	123.8	1409	12	0	0	9
BBC News	156.65	161.18	2.9%	127.8	1452	6	0	0	0
Amazon Kindle	201.93	207.74	2.9%	99.4	1180	22	0	0	0
Pedometer	207.84	212.57	2.3%	880.7	141	6	22744	0	0
Sensor Box	256.76	263.16	2.5%	1100.6	39	14	28656	19	0

We have measured the time overhead. Table 3.4 shows the result of 8 apps, which are either touch-intensive or sensor-intensive. Column 2 is the original run time, which is the time of an app running in the record mode. Column 3 is the replay time, which is the time of the app running in the replay mode with the corresponding recorded data. Column 4 is the time overhead. As we can see, the time overhead ranges roughly from 2% to 4%. We believe that at least the following three factors contribute to the overhead. First, during the replay, after injecting the first event of a pair of consecutive events, MobiPlay waits a period of the time interval of the two events before injecting the second event by utilizing the *thread.sleep* method; however, *thread.sleep* is inaccurate, and operation in parallel can lead to excessive sleep. Second, it takes time for MobiPlay to adjust the time information of an event on basis of current time. Third, reading the input data from the

disk needs time as well.

We also have measured the size of the recorded input data for the same 8 apps. Additionally, we have recorded the number of events for each of five categories. In Table 3.4, ME, KE, SE, R, and L stand for MotionEvent, KeyEvent, SensorEvent, Rotation, and Location, respectively. The first six are touch-intensive, and the rest two are sensor-intensive. As the table illustrates, the more events, the larger the data size; and each motion event occupies more space than each sensor event, since the former has more parameters than the latter.

### 3.4.4 Event Sampling



**Figure 3.8:** Re-sampling reduces the replay time.

Our evaluation here focuses on the event re-sampling on touchscreen gestures themselves, i.e, the MotionEvent. RERAN has conducted similar test, but it has focused on time warping during data entry (such as shrinking time interval between two button presses) and content processing (such as reading a story) instead of touchscreen gestures. RERAN states that manipulating the speed of touchscreen gestures can easily modify the gesture’s effect or convert it to a different action or set

of actions. We have found that our replay approach can speed up the touchscreen gestures without any error. Specifically, we sped up the touchscreen gestures twice. For instance, a *swipe* consists of an ACTION\_DOWN, a sequence of ACTION\_MOVE, and an ACTION\_UP. We shrunk the time interval between each pair of ACTION\_MOVE by half. Figure 3.8 shows the results of the same 8 apps in Section 3.4.3. The numbers above the re-sample column is the percentage of time that has been reduced by re-sampling. As we can see, the more the number of motion events, the more the time reduced (please refer to Table 3.4).

### 3.5 Limitations and Future Work

One limitation of MobiPlay is that the server in our current implementation is x86-based, preventing MobiPlay from running apps that need ARM-based third-party libraries, such as 3D apps. Fortunately, there are ARM-based servers available now, and both KVM [91] and Xen [36] offer extensions for ARM architecture. We leave the implementation of MobiPlay with an ARM-based server as our future work.

There are also several other directions for the future work. In principle, MobiPlay should be able to replay an app on a mobile device with the input data recorded from another device with distinct device configuration, such as resolution, screen size, etc. Therefore, one direction is to test and evaluate the cross-device portability on MobiPlay. In addition, it is worth improving MobiPlay such that it can support camera and microphone, which it currently does not.

### 3.6 Related Work

A large body of research has been conducted in record-and-replay techniques, including desktop, server, and mobile phone applications. In this section, we review the most relevant works from recent literature.

**Desktop and Server Applications** There are bunch of record-and-replay tools in the last decade [29, 30, 60, 84, 86, 88, 90, 105, 116, 123, 125, 140]. Among them, some are event-driven, such as [84, 123]; these tools record the (x,y) pixel coordinates of mouse clicks as well as keyboard

strokes, and replay this recorded information by creating new mouse and keyboard events later. Some systems utilize the keyword action technique, such as [29,105,125]; they work in a higher level of abstraction by capturing GUI objects. Even though some of these tools [84,88,123] record mouse move and mouse drag information, they cannot record and replay gestures on mobile phones like swipe, pinch and zoom due to the added complexity of multi-touch.

There are several other works in this line of research. [44] has presented a tool called Timelapse for quickly recording, reproducing, and debugging interactive behaviors in web applications. [55,71] have designed an approach for generating test cases for web service applications. [78] has presented an approach called PUPLE to provide automated support for capturing and replaying configuration decisions. [106] has presented a tool to learn how to interact with the application under testing and stimulate its functionality by working at the system level and interacting only through the GUI. And [149] presents a system for automating bug reproduction.

**Mobile Phone Applications** The android SDK offers a tool called Monkey [34], which can be run on any device or emulator instance. It can generate pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. Monkey supports event sequence scripts to be fed into an app. It can also handle presses but scripting presses is labor-intensive. Furthermore, Monkey scripting does not support touchscreen gestures.

Google has also provided several tools [63,115,126,146]. Monkeyrunner [115] provides an API for writing programs that control an Android device or emulator, and allows a developer to externally exercise an app. Robotium [126] fully supports native and hybrid Android apps and makes it easy to write automatic black-box tests. UI Automator [146] provides a set of APIs to build UI tests that perform interactions on user apps and system apps. Espresso [63] provides a set of APIs to test user flows within an app. All these tools hook into the app source code, which is a limitation as source code is not always available. There is another framework called GUITAR [73] for Java and Windows apps. It has been ported to Android by extending the Monkeyrunner tool to allow users to generate test cases. However, it does not support touchscreen gestures and many sensors typically found on mobile phones. MobiGUITAR [31] presents a tool for automated GUI-driven testing of Android apps, based on observation, extraction, and abstraction of the run-time state of

GUI widgets.

RERAN [67] provides a record-and-replay tool to capture low-level event streams on the mobile phone, including GUI events and sensor events. However, it is not able to record and replay data from the GPS and microphone devices, because Android provides data for these devices through specified services. Furthermore, it has a potential concern regarding time dependence of events. Finally, because of the design of the replay agent there may be conflicts with other events occurring at the same time. Mosaic [75] provides a virtual screen to handle the differences across different devices. It maps a set of touchscreen events from a particular device into a set of virtualized user interfaces that can be retargeted and injected into another device. Both RERAN and Mosaic utilize *getevent* to record app data and *sendevent* to inject the recorded data for app replay. Therefore, it retains some of the problems present in RERAN. Selendroid [137] presents a test framework based on Android instrumentation framework and good for white-box testing. However, it does not provide recording functionality.

Record and replay functionality can also aid security researchers. Work that makes use of physical sensors on the Android platform, such as microphones, accelerometers, and speakers for security purposes, such as [76,117,119,159], can benefit greatly from MobiPlay. Using our system, researchers can debug their applications in a traditional way. And, they can also perform sophisticated security analysis, such as monitoring sensor use, and exploring the feasibility of replay attacks.

### 3.7 Chapter Summary

In this chapter, we have designed a client-server system, called MobiPlay, which allows users to record and replay mobile application executions. MobiPlay runs the the target mobile application on a server, while displaying the app GUI in real time on the mobile phone, such that the mobile phone user has exactly the same experience as if the application were running on the mobile phone. We are the first to build such a system, which records the input data at the application layer, instead of the Android framework, or the Linux kernel. This allows us to solve many difficulties the current approaches have encountered.

MobiPlay is comprehensive, flexible and efficient. First, it is able to intercept all input data,

including all touchscreen gestures, and data from all sensors, better than current state-of-the-art approaches. Second, it is able to record and replay a mobile app on both the mobile phone or the server. Third, it is suitable for both white-box and black-box testing. We have implemented MobiPlay on Android and evaluated it with tens of popular applications, with supportive and convincing results.

## Chapter 4

# vRENT: Virtual Machine Migration on the Pervasive Edge for IoT Applications

### 4.1 Introduction

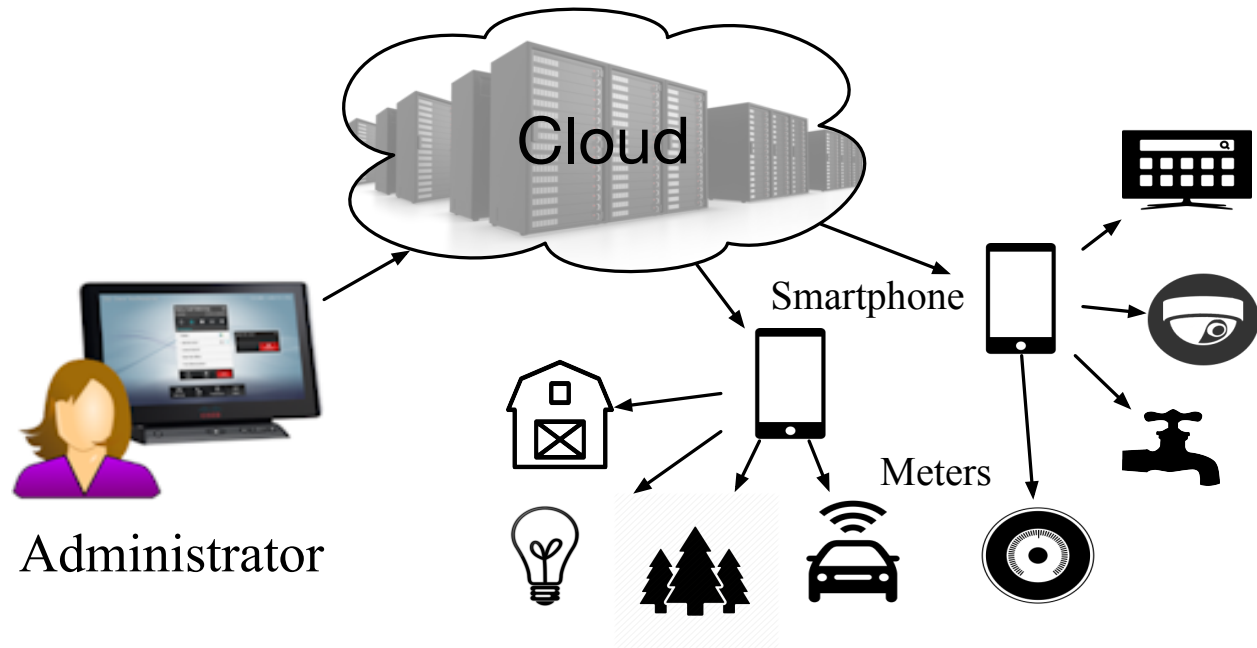
Nowadays, smartphones have become more and more popular and powerful, with impressive advantages such as mobile coverage, rich resources, and reliable network connectivity. It would greatly benefit the IoT if such advantages can be brought in. For example, there is already a trend of merging IoT and smartphones in Android Things in which IoT devices have similar architecture as smartphones [69]. We have also seen the computation offloading pattern between smartphones and smartwatches/smartbands, in which smartphones take over heavy computation tasks from smartwatches/smartbands [64, 82, 122].

Our motivation is that computing resources of smartphones are not fully utilized all the time. Hence, it is possible to use an incentive mechanism to recruit smartphones as edge nodes to complement or substitute the fixed sink nodes in current IoT systems. Different from fixed sink nodes, smartphones can collect data from smart meters along the paths of smartphone owners, enhancing the mobility of the IoT network. They can also process data at a very early stage, cutting the response delay for IoT applications. Furthermore, they can help scale up IoT easily at low cost. An illustrative example would be a smart city with massive smart meters deployed at various locations. As mobile users come and go, their smartphones can collect data from smart meters within the



communication range. At the same time, smartphones can help process the collected data, store it, and send it to backend servers or cloud services.

In general, our system consists of four entities: *third party application (administrator)*, *cloud service*, *smartphone edge nodes*, and *meter*, as shown in Figure 4.1. The third party issues an



**Figure 4.1:** The scenario where the smartphone serves as the hub of IoT networks.

application request to the cloud service, which relays the request to a set of chosen smartphone edge nodes. After receiving the request, each smartphone allocates resources to run the application. The application can interact with the nearby meters through wireless links, such as collecting data from meters, processing data, and sending feedback or control command back to meters. The application can also send pre-processed data back to the third party through the cloud service.

There are two major challenges to realize the above system. The first challenge is the security. From either the perspective of the smartphone or that of the IoT application, the other entity is untrusted. On one hand, although the owner of the smartphone is willing to sell her idle resources, she must be guaranteed that there is no security and privacy risk or hardware/software damage to her own device. On the other hand, the IoT system must be assured that the smartphone keeps the integrity of its data, processes the data as it is instructed to (no unauthorized data mining), and

does not leak any content of the application. The second challenge is the continuity of application execution. A smartphone may abort the execution of the application, due to either the smart phone is out of the communication range of the meters or its available resources run out. In these cases, the smartphone must migrate the unfinished tasks to another nearby smartphone, or hand it over to a normally deployed sink node, or upload it to cloud services for further processing.

In addressing these challenges, we present vRENT, a new mechanism to manage device resource of smartphone based on Xen virtualization and MiniOS. vRENT enforce isolation and security by elevating user's Android OS as a Guest OS and renting smartphone's resource in the form of MiniOS. Specifically, in vRENT, the hypervisor and Dom0 manages all resources. The smartphone users and renters can only access the resources in the Guest OS and MiniOS respectively, they cannot access any resource beyond their domains. In addition, vRENT presents an effective and efficient scheme for live MiniOS migration, which allows unfinished tasks running in the MiniOS to be migrated to other entities when needed.

The benefits provided by vRENT are prominent. In term of security, vRENT sandboxes the MiniOS and Guest OS in two different domains and prevents them from accessing the resource of each other. Thus, the smartphone owners do not need to worry about their software/hardware being harmed from renting their idle resource, and renters are willing to run high assurance tasks on the rented resource. Furthermore, vRENT can easily manage the rented resource. By renting resource in the form of MiniOS and supporting live migration, renters are more flexible for designing their applications. They can manage and access all resource of MiniOS, and decide when to start, stop and migrate the tasks as they want.

Comparing to traditional migration methods [20, 40, 41, 54, 87, 131, 133], vRENT is more flexible and efficient. Traditional migration methods rely on hypervisor to save system states. During migration, traditional methods usually require hypervisor to over-conservatively collect all system states and user data, leading to massive image volume. Even though they are useful for PCs and cloud servers since they have more resources, they are inefficient in our renting scenarios. On the contrary, vRENT allows the MiniOS to save system states, giving MiniOS more freedom since it can decide when to migrate without fully shutting down the MiniOS. Meanwhile, vRENT can smartly

locate the useful data that must be migrated, usually generating far less data than traditional migration methods.

In summary, our main contributions are as follows:

- We present a novel virtualization and MiniOS based mechanism – vRENT, which allows smart-phone users to safely rent their idle resource to third party renters for managing their IoT devices.
- We design an efficient and effective migration method for renter to live migrate their MiniOS to other devices.
- We have implemented a prototype system on the development board and conducted extensive experiments to validate the design.

The rest of the paper is organized as follows. We review the related work in §4.2, and present the overview of vRENT in §4.3. §4.4 details the migration scheme of vRENT. §4.5 offers the implementation of our prototype system in detail. We present the evaluation in §4.6 and conclude the paper in §4.8.

## 4.2 Related Work

Nowadays, smartphones have been increasingly integrated into Internet of Things (IoT) that interweaves ubiquitous computing, wireless communications, and edge/cloud computing. In this section, we will briefly review the IoT application and its relationship to smartphone and point out the difference between our work and the existing work. As the focus of our work is on application migration on smartphones, we will also review work along this line.

### 4.2.1 IoT Application

Due to the dominance of cloud computing, there is massive adoption of cloud platform in IoT applications [32,68,83,112]. Beside the cloud platforms, there are also many embedded platforms(e.g. smartphones, mini PC, and micro-controllers, etc) can be alternative gateway or hub of IoT [161]. An IoT application usually needs to address data collection, data analytics, and data management.

Data collection needs to incorporate the sensors on the hub/board or wireless sensors scattered. The hub node need to be carefully placed to provide best coverage to those sensor nodes. The situation get worse when the sensor nodes are mobile nodes moving all the time, which make always-on always-connected cloud-based solution popular. Depending on various latency goals, quality goals and the sizes of data volume, the data analytics can be hold or partially hold on the sensor node, or on the sensor hub, or in the cloud. The data management needs to provide efficient data store, share, and search, which can be seen mostly on the sensor hub or cloud node [74]. Admittedly, cloud can provide holistic solution to data collection, data analytics, and data management. However, the cloud is not suitable for addressing IoT applications that have human in the loop or mission-critic application with low-latency requirements. Such requirements make cloud-based solution less-favored due to its unpredictable performance.

#### **4.2.2 Smartphone in IoT**

With more and more powerful hardware, smartphones have been playing important roles in IoT applications. For instance, they have been utilized for data collection and data relay in wireless networks, such as work [99, 121, 155]. Furthermore, they have been harnessed in IoT-related environments in work [160], which points out that the Internet of Things has a gateway problem and smartphones can be served as gateways to connect IoT peripherals and cloud services. Our system shares stark difference as the smartphones in our system are not trusted. Due to the mobility and serving purpose, smartphones are allowed to join or leave the IoT system at will. Therefore, with our system, IoT devices are not necessarily bounded to a single smartphone that has to be owned by the same person. Most importantly, our system solves the problem of continuum execution by migrating the unfinished application seamlessly to other smartphones.

#### **4.2.3 Edge computing**

Edge computing is a revolutionary concept that provides elastic resources such as computation, networking, memory and storage at the edge of networks rather than in the core of networks [39, 133, 139, 157]. Edge computing has been considered as the engine of IoT, and various work has

demonstrated the advantages in using edge computing as the IoT service endpoint [135]. Researchers have investigated the design of edge computing node [102, 134, 151], the components of an edge computing node [156], programming model [77, 80], and security & privacy issues [94, 158]. Our work is related to edge computing node design, which is different from existing work. Cloudlet [134] is built on high-end server machines using OpenStack techniques. ParaDrop [151] is implemented on wireless router using OS-level virtualization. Our edge node design focuses more on the mobile side. We are the first to consider edge node design and implementation on smartphones. Additionally, our work has explored a new edge computing deployment model with new constraints. Different from the existing work where edge nodes are assumed to belong the same organization, smartphones in our system are recruited from general mobile users.

#### 4.2.4 Virtualization and Migration

Some works attempt to provide a secure and efficient execution environment with the aid of Xen and MiniOS. ClickOS [107] optimizes the MiniOS and Xen I/O subsystem to provide high efficient middleboxes. Unikernel [104] provides a new approach to deploy cloud services via single-purpose applications running in MiniOS. Unfortunately, these work focus on high performed x86-based cloud servers rather than resource-limited ARM-based smartphones. Jitsu [103] present a new mechanism for securely managing multi-tenant network apps on ARM architecture. It optimizes the Xen toolstack to achieve fast boot and efficient communication. However, this work does not take the migration into consideration, thus it cannot be applied to our scenarios. As to migration, most of existing work focuses on full operating system migration on x86 [40, 41, 41, 54, 87, 131, 133], either on commodity computers or on servers. The only project that support MiniOS migration is MirageOS [20], however, this migration mechanism heavily relies on Xen hypervisor or Dom0 to manage and transfer, which is not suitable to our scenarios.

### 4.3 Problem and Overview

In this section, we provide an overview of vRENT. We first introduce the preliminaries and our design goals in §4.3.1 and §4.3.2, respectively, then describe how vRENT works in §4.3.3, and finally

presents the assumptions in §4.3.4.

### 4.3.1 Preliminaries

The ARM-powered edge device is the main focus of this paper, and the design and implementation on its side is, in short, a MiniOS on Xen hypervisor on top of ARM architecture. Thus, we briefly describes Xen for ARM, and MiniOS here.

- **Xen for ARM.** Xen [36] is a widely used virtualization technique that isolates multiple virtual machines and enables them to run on top of a shared hardware resource. It was originally developed for x86-based hardware. Starting from Xen 4.4, it has released additional support for ARM architectures, specifically ARM v7-A and ARM v8-A, including extensions that let a hypervisor manage hardware virtualized guests without the complexity of full paravirtualization. Compared with x86, the Xen for ARM port is much simpler since it can avoid many legacy requirements such as Qemu device emulation. ARM virtualization extensions are a great fit for the Xen architecture. In Xen for ARM, Xen runs entirely and only in hypervisor mode, which significantly reduces the number of context switches required. In addition, Xen for ARM uses 2-stage translation in the MMU to assign memory to virtual machines, such that it is convenient and efficient for Xen to manage VM memory.
- **MiniOS.** MiniOS is a small OS kernel distributed with the Xen hypervisor sources. It has been used as a basis for development of Unikernels, such as ClickOS [107] and MirageOS [104]. MiniOS only provides the basic functionalities required to run a Xen virtual machine, such as providing code to initialize the CPU, displaying message on the console, allocating memory, etc. To reduce the cost of context switches, MiniOS runs all processes in the kernel mode. The processes are scheduled cooperatively, using a round-robin scheduling algorithm. According to this algorithm, a process yields the control of the processor to another process when a blocking function like *msleep* is called or when an interrupt happens.

### 4.3.2 Design Goals

vRENT seeks to achieve the following objectives:

- **Strong Isolation.** vRENT should guarantee that all processes in the user’s OS cannot access the resources rented to third party and vice versa.
- **Reliable Resource.** vRENT should provides stable resources to third-party renters. During the lease term, all rented resources should be fully controlled by third-party renters.
- **Easy Management.** vRENT should offer smartphone owners a convenient management interface for deciding whether the owners are willing to rent their idle resources to third parties.
- **Low Cost Migration.** When third-party renters need to migrate their processes to other entities, vRENT should provide them an efficient migration mechanism with low cost. In other words, the migration procedure has to be light-weight enough and does not consume too much resource.

### 4.3.3 vRENT Overview

An overview of vRENT is presented in Figure 4.2. At a high level, it consists of four key components: *cloud service* running in the cloud, *Dom0*, *Android OS*, and *MiniOS* running in the smartphone. The cloud server maintains the list of registered smartphones and dispatches MiniOS images to these smartphones when needed. Dom0 can be seen as a part of hypervisor, which is in charge of creating the MiniOS and communicating with the cloud service. MiniOS image encapsulates the IoT application. Note that MiniOS is provided by the third party renter and implemented on smartphones to manage nearby meters. Android OS is the only component that smartphone owners can access, and it is provided with a client application to register the smartphone to the cloud service for sharing its resource. As a guest OS in Xen, Android OS cannot access the resource in other domains such as Dom0 and MiniOS. In the following steps, we use an example to show how vRENT works when receiving a resource sharing request.

1. The third party renter designs the MiniOS image for managing meters and sends it to the cloud service (e.g. an image registry service);

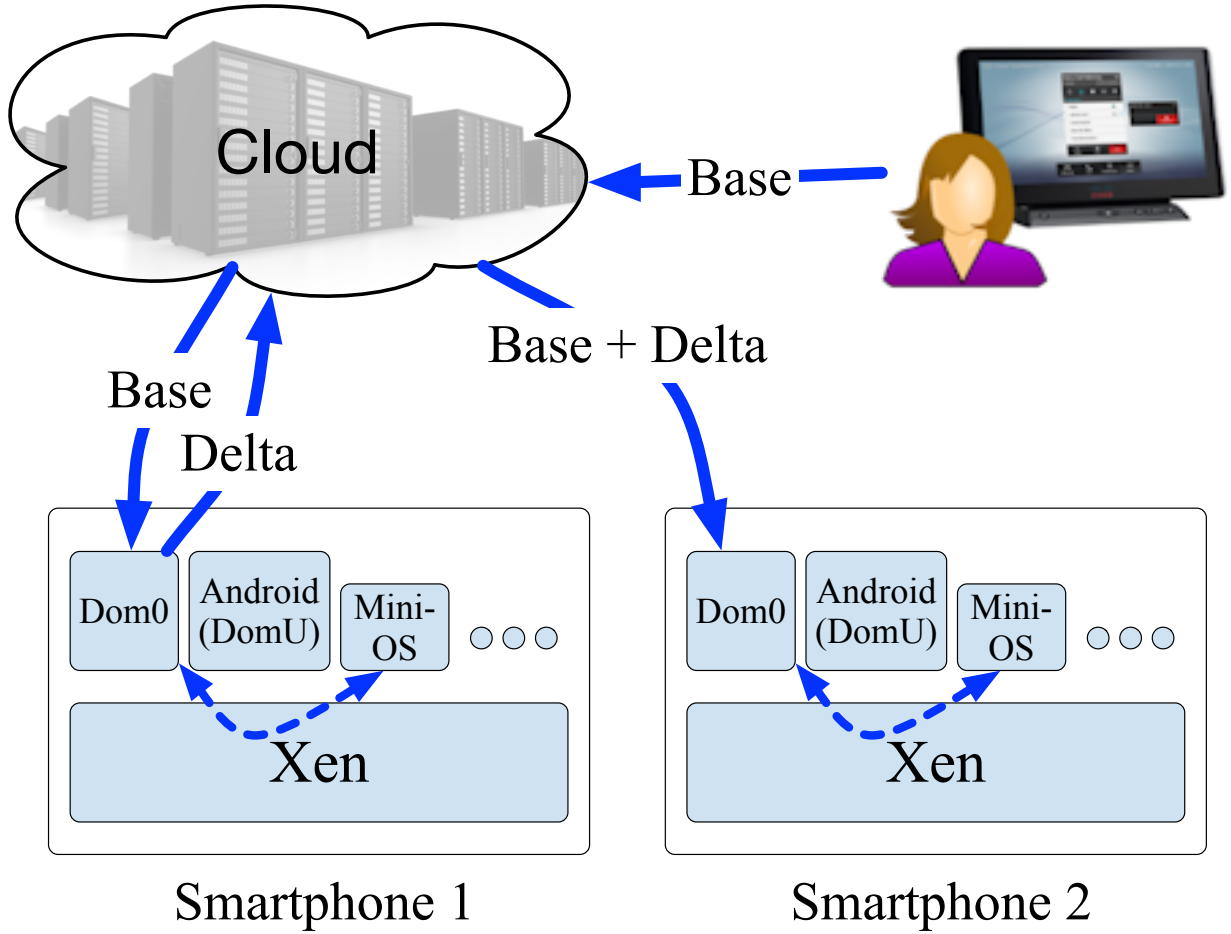
2. The cloud service looks through all registered smartphones and relays the MiniOS image to Dom0 of the smartphone that is near to the target meter;
3. At the smartphone side, the Dom0 creates a MiniOS instance with the given image;
4. The created MiniOS starts the application and executes its tasks;
5. If MiniOS has finished all the tasks of the application, Dom0 revokes all resources and sends the result of the application back to the cloud service;
6. If the smartphone cannot continue to provide resources while the tasks of the application are not finished yet, MiniOS creates an migration image of the unfinished tasks. Then, Dom0 revokes all resources and sends the migration image back to the cloud service;
7. The cloud service sends the result to the third party renter if the application is finished, or finds another available smartphone to continue the unfinished application by sending it the original MiniOS image and the migration image.

From the above example, we can see that the key enabling techniques of vRENT are the virtualization and MiniOS migration. vRENT leverages Xen for ARM to provide virtualization and MiniOS. In the meanwhile, vRENT presents a novel method for MiniOS migration, which will be detailed in §4.4

#### 4.3.4 Assumptions

Before we start presenting the migration detail, we first define some assumptions. In our system, we focus on the security concern of the smartphone and the IoT application, instead of other components and communication links. Thus, we assume that the cloud service is trusted, which is usually the case for large commercial cloud services like Amazon EC2, Microsoft Azure, and Google Cloud Platform. We also assume that all the communication links are secure due to standard secure network protocols. We further trust the meters themselves, since they are deployed, controlled and configured by the Third-party(application administrator).





**Figure 4.2:** The migration procedure.

We also assume that, motivated by certain incentive mechanism, the smartphone is willing to rent a portion of its resource. We assume that the virtualization environment is deployed by the vendor (like Sumsung) as a one time effort before the smartpone is released to the market.

We finally assume that the vendor provides a pre-installed application (an Android OS) that allows the mobile owner to decide how much resource she is willing to rent, and the mobile owner only has access to that Android OS. All the activities of hypervisor, including Dom0) and MiniOS, run in the background and are transparent to the mobile owner. As a result, processes running in the Android OS have no access to the resource of the hypervisor or MiniOS.

## 4.4 Migration Procedure

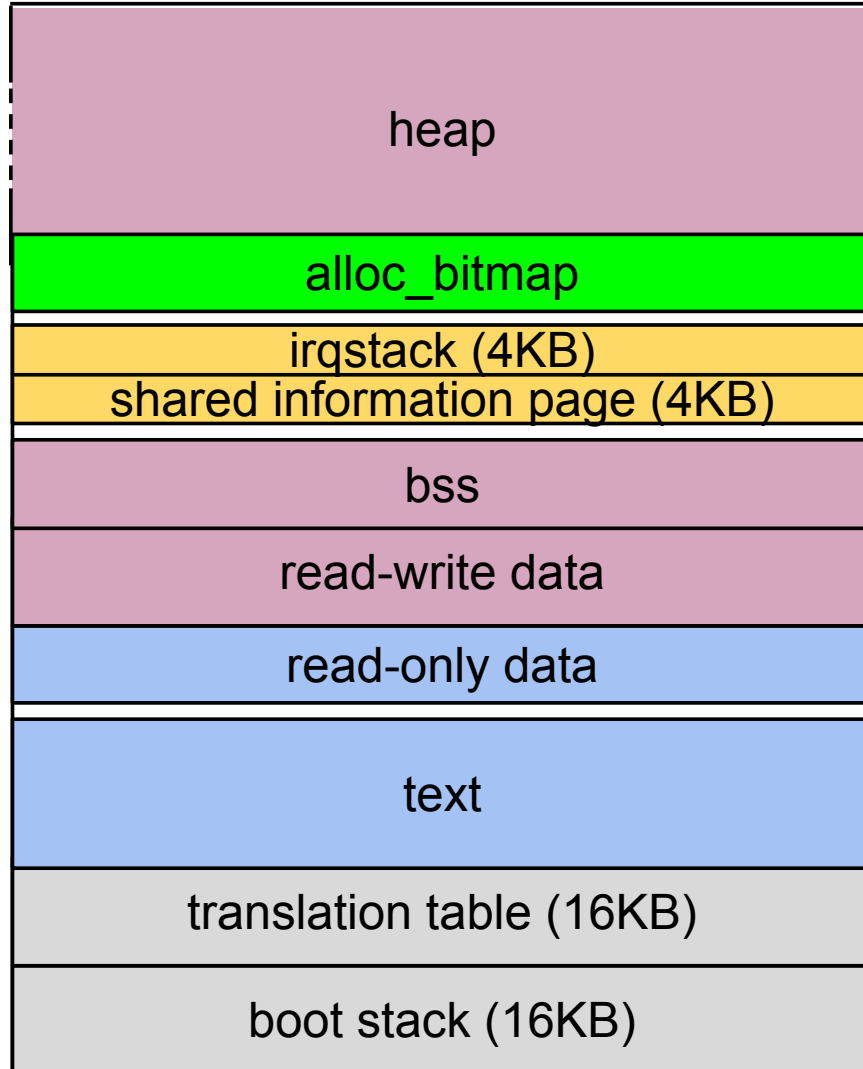
The most important advantage of vRENT is the flexibility of smartphone participation, which allows smartphones join and leave the IoT system at any moment. It is inevitable that a smartphone may leave the system before the finish of tasks of an application. For instance, the smartphone is far away from the meters related to the application, or the smartphone decides to stop providing its resources. As a result, the unfinished application must be picked up by another smartphone seamlessly. In the following, we will detail how to migrate the unfinished application from one smartphone to another.

### 4.4.1 What to Migrate

To migrate the unfinished application as quickly as possible, we first must determine the only necessary data that needs to be migrated. The data can be categorized to memory data, CPU states, and disk data. *Memory data* includes memory allocated by application processes, such as stacks and heaps, and memory allocated by system processes and drivers for communication between MiniOS and Dom0, such as ring buffers. *CPU state* is a set of registers that records the status of current running processes, which is stored into memory whenever a context switch happens. *Disk data* refers the data that is exclusively for the application. Since the disk migration is relatively easy, just by copying all data in source disk to the destination disk, and CPU state will stored into the memory when the migration process runs, our problem boils down to migrate the memory data.

Next we need to analyze the memory usage in MiniOS. The virtual memory address space is shown Figure 4.3, and we can classify the memory into three categories:

1. The memory that is not modified during system running such as the *read-only data* and the *text* segment, and that is used by by system processes and system structures to support system running such as the *boot stack* segment and the *translation table* segment;
2. The memory that is used by drivers or system processes for communication between MiniOS and Xen, such as the *shared information page* segment. This category of memory varies among different devices,allocated mainly by *malloc* function and partially by pre-allocation when system boots such as the shared information page segment.



**Figure 4.3:** Virtual address space.

3. The memory that is used by application processes, including all application data, some system data in the *read-write data* and the *bss* segment, and all memories allocated by *malloc* function by application processes.

We divide the MiniOS virtual machine into two parts: *Base* and *Delta*. *Base* contains resources that do not change, and *Delta* contains resources that change as the application is running. We observe that the virtual memory distribution is always the same when MiniOS boots to the point right before the application processes are created. During each boot, MiniOS initializes resources

like CPU, memory and all data structures, and creates several system processes such as idle process and *xen\_store* process. Thus, we take all resources allocated before the creation of application processes as Base, and all resources allocated thereafter as Delta.

It is clear that the memory of category 1 belongs to Base and that of category 3 belongs to Delta. However, it is not easy to infer which the memory of category 2 belongs. For instance, when an application process in MiniOS wants to send out a packet via network, it will first call the networking API, such as *netconn\_write*, which then transmits the packet to Dom0 via *netfront driver*. Once Dom0 sends out the packet, it will send an acknowledge back to MiniOS. If the migration starts before the acknowledge returns, it may cause a problem, because once it is migrated to the destination smartphone, it will keep waiting for the acknowledge that never comes. The reason is that each driver has its own data structure to keep device-specified information, which contains a field as a pointer to ring buffer shared with Xen hypervisor. When the system boots in destination smartphone, it has to recreate those shared ring buffers. As these ring buffers may reside at different addresses, we have to reassign their addresses to the pointers in driver data structure. However, this cannot be done without analyzing the application’s source code, which violates our objective of designing a general system for fast migration. Therefore, we design to shut off all device drivers before migration and initialize them in the destination smartphone before resuming the application. And we require the application processes access drivers only by APIs provided by vRENT. In this way, the memory of category 2 does not need to be migrated, since it is freed before migration. Thus, our work boils further down to migrate the memory of category 3.

We conduct a thorough research on the source code of MiniOS, and find that the following five data structures in the memory of category 3 must be migrated:

- ***freelist***. *freelist* is the head of a link list that records all free memories less than 1 page size. When a process calls *malloc* function, it will first check if there is an element in *freelist* satisfying the size requirement, otherwise turns to the memory chunk. Freelist is used to avoids too many small memory chunks and is very efficient for small-size memory allocation.
- ***free\_head*** and ***free\_tail***. These two structures are arrays of link lists maintaining the locations of a certain number of free pages in MiniOS. Each element in the array keeps the

address of a certain number of free pages. For instance, in the array for 2 free pages, each element in the array stores an address of a two-page free memory. As a result, MiniOS can quickly allocate the most suitable memory chunk to respond to a memory request. For example, if the request is a 1.5 pages, MiniOS will find a memory chunk in the 2-page array, and put the half page unused into the *freelist*.

- ***alloc\_bitmap***. This structure is a pointer to an array that MiniOS uses to record whether a page in the heap segment is allocated or not. Each bit in the array corresponds to a page. For example, `alloc_bitmap[0] = 5` represents that page 0 and page 2 are not allocated. This array is stored in an address space just below the heap segment as shown in Figure 4.3.
- ***thread\_list***. *thread\_list* is the head of link list maintained by CPU scheduler to record all running threads in the system. Each element of this link list is a data structure that records a thread's information. Almost all scheduling operations are associated with *thread\_list*. For example, Whenever a thread is created or deleted, it will be added into or removed from the link list by the system.
- ***exited\_thread***. *exited\_thread* is used to record all threads that have finished their jobs. The scheduler will remove these threads from the system in context switch, and release the resources allocated to them.

All these five data structures are declared as static variables and stored in the *bss* segment. The first three are memory-related data structures that record the distribution of the *heap* segment, and the last two are used by the scheduler to manage all processes.

#### 4.4.2 How to Migrate

Once we know the concept of Delta for migration, the next is how to migrate it. The most important challenges are:

#### 4.4.2.1 How to design the migration process

We need a dedicated process to handle the migration. Since this process runs as a standard process in MiniOS, it also consumes memory resource, which could be mingled together with the memory chunks which should be migrated. Not only should it avoid migrating the memory allocated by its own to the destination device, but also protect itself from affecting critical system data structures like *freelist*. Furthermore, MiniOS has no user/kernel space separation and all processes run in the kernel mode with a cooperative scheduler. Therefore, the migration process cannot be assigned a higher priority than others, and it can be interrupted by an *irq* (interrupt request), which may trigger the scheduler to run another process. During migration, if another process gets the CPU, it may allocate new memory resource and destroy the memory distribution which we are migrating.

#### 4.4.2.2 Where to save Delta

As we have mentioned before, we shut off all devices used by the application processes, including networking. Therefore, we cannot transmit Delta to the cloud service during migration. We cannot save Delta into memory either, since the smartphone may have limited memory resource. As a result, we need a storage space which is large enough to be used exclusively by the migration process.

#### 4.4.2.3 How to locate Delta

The memory resource owned by application processes with five system data structures discussed above are located in different memory address spaces (*bss*, *heap*, and *read-write data*) and mixed with data that does not need to be migrated. We design a scheme to locate the data indispensable for migration and we explain it in the following steps:

- (I) **Allocate Memory Early.** We design a system process called migration to handle the MiniOS migration. MiniOS utilizes migration and three other processes, *xenstore*, *shutdown* and *idle*, to support system running. As mentioned before, MiniOS keeps three data structures to manage memory in the *heap* segment. Unlike memory in other segments like the *bss* segment where each address is statically allocated to processes, memory in heap segment is dynamically allocated to processes by *malloc* or *malloc*-like functions. To minimize the size of Delta, it

would be worth separating system process memory from application memory in the *heap* segment and only migrate the latter.

To begin with, we first need to understand how the four system processes use the *heap* segment. *xenstore* manages all communications with Dom0, receiving data from or transmitting data to Dom0 leveraging shared queues. It allocates new heap memory when the shared queues are not empty. Otherwise it blocks itself and does not allocate new heap memory. *shutdown* monitors the shutdown request from Dom0 by checking the “control/shutdown” key in *xenstore*; whenever a change is made on this key (usually by Dom0), *shutdown* will execute the command indicated by the key (power off or reboot). and it does not allocate new heap memory. *idle* starts to run if no other runnable thread can be scheduled on CPU, and it does not allocate new heap memory. *migration* is provided by *vRENT* to implement all migration-related operations, and it only consumes heap memory by calling *malloc* function. In summary, *shutdown* and *idle* do not allocate new heap memory. *migration* always allocates new heap memory, and *xenstore* allocates new heap memory when its shared queues are not empty.

Therefore, we design the system as follows. First, before running application processes, we create all system processes and pre-allocate enough memory to migration process such that the system processes have the same memory distribution for every boot and migration process does not need to allocate memory later; that is, the system processes have stable memory resource, which will not be taken by migration process later. Second, if we plan to obtain a memory snapshot, we do so only when *xenstore* has empty buffers to make sure no new heap memory allocated.

- (II) **Save Data to Disk.** We design to store Delta to disk. We create an extra para-virtualized disk for MiniOS, called *Disk0*, to exclusively save the migration data, and require the user to include *Disk0* in the configuration file.
- (III) **Mine Necessary Data for Migration.** As discussed before, when the application starts to run, the modified memory resources are the application memory and the five data structures. To obtain the modified memory, we keep two snapshots of the selected memory at different time

including the *read-write data*, the *bss* segment and *alloc\_bitmap*. The first snapshot, called *snapshot0*, is obtained at the point right before the application processes are created, and the second, called *snapshot1*, is obtained at the time when the migration begins. Specifically, when the migration process decides to obtain a snapshot, MiniOS will first shut off all drivers and stop the Xen bus communication, such as invalidating the data used to communicate with Dom0, and then get the snapshot and store them to the memory space of the migration process. Finally, the migration process performs XOR operation on the two snapshots, and then save the XORed results into *Disk0*; the XORed result of *alloc\_bitmap* indicates which pages in heap segment contain the application data, thus we store these pages into *Disk0* as well. We also save the current *freelist*, *free\_head* and *free\_tail* data structures to *Disk0*. In this way, we can separate the application data from the system data, and it is also convenient to resume data in another smartphone.

- (IV) **Use CPU Exclusively.** When the migration process performs MiniOS migration, it has to keep the memory distribution unchanged. Therefore, we design to stop all other processes from executing (due to their execution can modify the memory distribution) with the following two methods: disabling *irq* and modifying *thread\_list*. When *irq* is disabled, the migration process does not yield the CPU even when an interrupt happens, which helps obtain memory snapshots we want. After obtaining snapshots, vRENT should resume the system processes and store Delta to *Disk0*, while still halting all application processes. Thus, we break up *thread\_list* link into two parts. The first part is for system processes that are still linked in *thread\_list*, and the second is for all application processes. We use a new data structure *migration\_thread\_list* as the head to maintain them. When MiniOS is migrated to another smartphone, these two link lists are merged into one such that all application processes can be resumed.

## 4.5 Implementation

To validate vRENT design, we have implemented a prototype system on top of Arndale Board-K development board, which is same as the boards used in Nexus 10 and Chromebook. This board



has two advantages for implementing our prototype system. One is that it has an ARM Cortex-A15 processor, which provides virtualization extension and is officially supported by Xen hypervisor. The other is that Linaro [23] provides open-sourced bootloader of this board.

#### 4.5.1 MiniOS Porting

Although Arndale development board is supported by Xen hypervisor, it is not yet supported by MiniOS. Fortunately, there is a version of MiniOS that works on another ARM based board called Cubieboard2 [24]. However, it cannot run directly on Arndale board. We find that in this version, dCache and the hardware float point bit cause the boot failure. After disabling them, we have managed to port the MiniOS.

#### 4.5.2 Device APIs

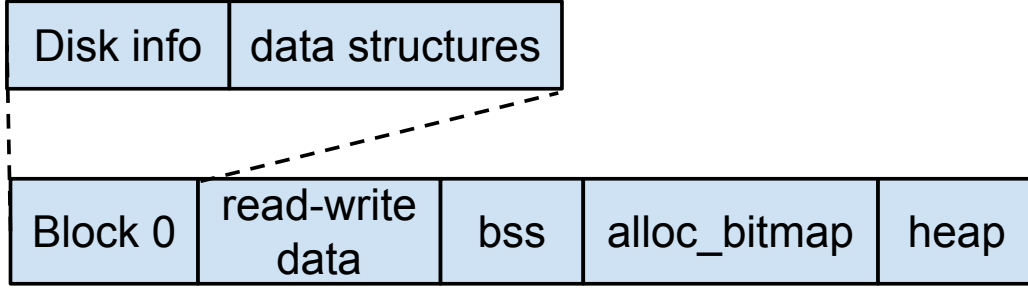
vRENT provides the following device supports:

**Network.** Network is required for communication with the cloud service, such as uploading Delta, downloading Base and/or Delta. In vRENT, we choose lwIP (lightweight IP) as the TCP/IP stack. lwIP is an open-source project designed for embedded system [25]. Occupying only a very small memory footprint (512KB), lwIP can provide applications the basic network access functionalities.

**Disk.** We provide APIs for the application to access disk, instead of providing a file system, because there is only one application in MiniOS. vRENT needs two disks, one for the application itself (*Disk1*)<sup>1</sup>, the other for the migration data (*Disk0*). The basic unit in disk is a 512 byte block. For *Disk1*, the application manages all these blocks and is aware of the data arrangement in each block. For *Disk0*, which is utilized to store the Delta data, we divide it into five sections (i.e. Block 0, the read-write data, the bss segment, the alloc\_bitmap segment, and the heap segment) as shown in Figure 4.4. Each section starts with a new block, and Block 0 keeps the disk information including whether this disk is empty, the start point, the length of each section, and the values of the system data structures (e.g. *freelist*, *free\_head*, *free\_tail*).

---

<sup>1</sup>Please refer to *Disk data* in Section 4.4.1



**Figure 4.4:** The layout of Disk0.

**GPS.** In vRENT, GPS is used to measure the distance between the smartphone and the meters and thus to trigger the migration when the smartphone is out of the communication range of the meters. The Arndale development board, however, does not have a GPS module. We hence design a GPS emulator to generate GPS data, which consists of three components: a generator, a backend driver, and a frontend driver. Particularly, the generator emulates the GPS driver to generate GPS data; the backend driver, located in Dom0, writes the data to the shared ring buffers; and the frontend driver, located in MiniOS, reads the data from the shared ring buffers and gives it to the application process.

**Bluetooth.** Bluetooth is used by the application to communicate with the meters. As the Arndale development board does not have a bluetooth module, we emulate the bluetooth device, similar as GPS.

We have implemented APIs for these devices, as listed in Table 4.1. Note that all devices have only *read* and *write* APIs, except that the network uses APIs provided by lwIP library. In this way, we can hide the low level details from the application and make it easier to implement the migration.

**Table 4.1:** The APIs for network, disk, GPS, and Bluetooth.

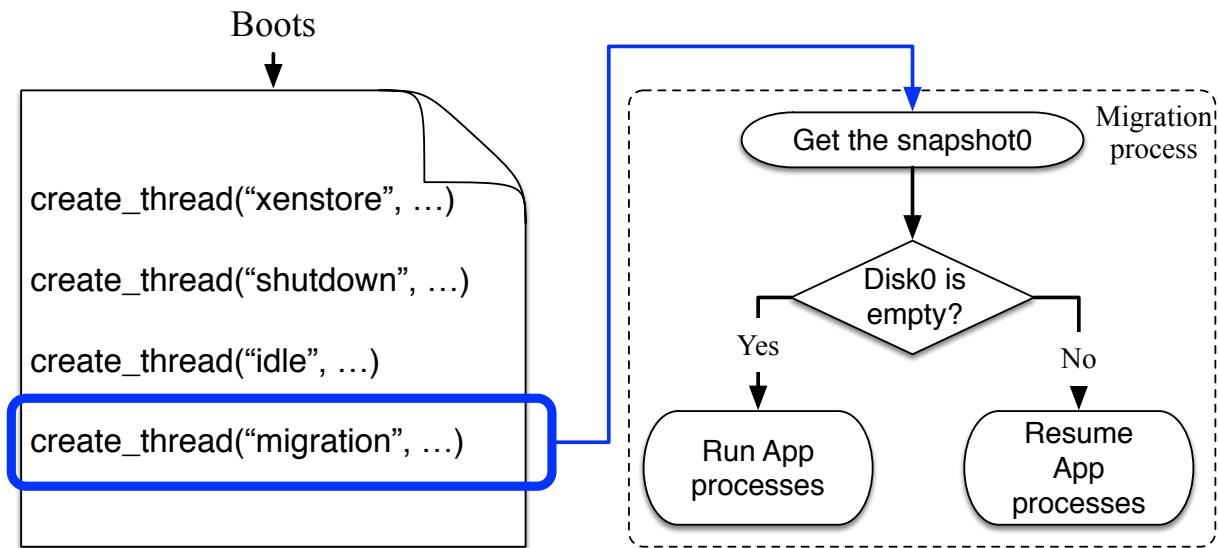
Device	API
Network	lwIP
Disk	int blk_read_sector(uint64_t sector, uint8_t *buf)
	int blk_write_sector(uint64_t sector, uint8_t *buf)
GPS	int gps_read(uint8_t *buf)
Bluetooth	int bth_read(uint8_t *buf)
	int bth_write(uint8_t *buf)

### 4.5.3 Application Programming

The application running in MiniOS is required to be compiled into a user library and uses *main\_app* as the entry point. It can only utilize APIs listed in Table 4.1 to access devices, since it is forbidden to modify the system memory. Furthermore, it must provide a network reconnection function in case that a new IP address is assigned to its MiniOS after being migrated to a new smartphone.

### 4.5.4 Migration Implementation

When MiniOS starts to boot, it first initializes all system resources. Then, as shown in Figure 4.5, MiniOS creates four system processes: *xenstore*, *shutdown*, *idle*, and *migration*. *migration* is created earlier than any of the application processes and serves as the entry point of the application code. Furthermore, it calls the *malloc* function only once to allocate memory large enough to finish all operations.



**Figure 4.5:** The migration implementation.

In a big picture, the migration process has two steps: *Save Delta* and *Restore Application*, which are detailed in algorithm 4 and algorithm 5, respectively.

---

**ALGORITHM 4:** The migration process: *Save Delta*.

---

```
/* The migration process does the following to obtain Delta and save it to Disk0 */
if xenstore buffer is not empty then
    | wait until xenstore buffer is empty;
end
disable irq to exclusively occupy CPU;
move all application processes from thread_list to migration_thread_list ;
enable irq and close drivers;
disable irq;
obtain snapshot1 and current value of freelist, free_head, free_tail and exited_thread;
obtain Delta by XORing on snapshot0 (please refer to Section 4.4.2) and snapshot1;
enable irq and open block driver;
store Delta, freelist, free_head, free_tail, exited_thread and migration_thread_list into
    Disk0;
for each bit of XORed alloc_bitmap, if it is 1, store the corresponding page into Disk0;
```

---

---

**ALGORITHM 5:** The migration process: *Restore Application*.

---

```
/* The migration process does the following to restore application */
read XORed read-write data, bss, alloc_bitmap and system data structure including freelist,
    free_head, free_tail, exited_thread and migration_thread_list from Disk0 to memory;
close all drivers;
wait until xenstore buffer is empty;
disable irq interrupt;
Resume read-write data, bss, alloc_bitmap by XORing on snapshot0 and data mentioned in
    step 1;
Resume freelist, free_head, free_tail ;
enable irq;
open all drivers except network driver;
for each bit of XORed alloc_bitmap, if it is 1, resume the corresponding page from Disk0;
disable irq;
merge migration_thread_list with thread_list and resume exited_thread;
enable irq;
```

---

## 4.6 Evaluation

In this section, we will evaluate our prototype system, including environment setup, boot time, save time, resume time, etc.

### 4.6.1 Environment Setup

The smartphone board in our prototype system is the Arndale Board-K development board. The board has the Exynos 5 Dual SoC, 1 GB RAM, and a 64 GB external SD card. It is also equipped with a 1.7 GHz dual-core ARM Cortex-A15 processor that provides hardware virtualization support. The cloud is emulated by a Lenovo Y480, equipped with a 2.4G Hz I7 processor, 8G memory, and a Qualcomm Atheros AR8161 gigabit Ethernet. The development board and the cloud are wired through a router. We use Xen4.4.1 for the hypervisor, u-boot from Linaro for the bootloader, and Linux 3.19.7 for Dom0 and Dom1. The application, i.e., MiniOS image, is provided to Dom0 by the cloud.

### 4.6.2 Boot Time

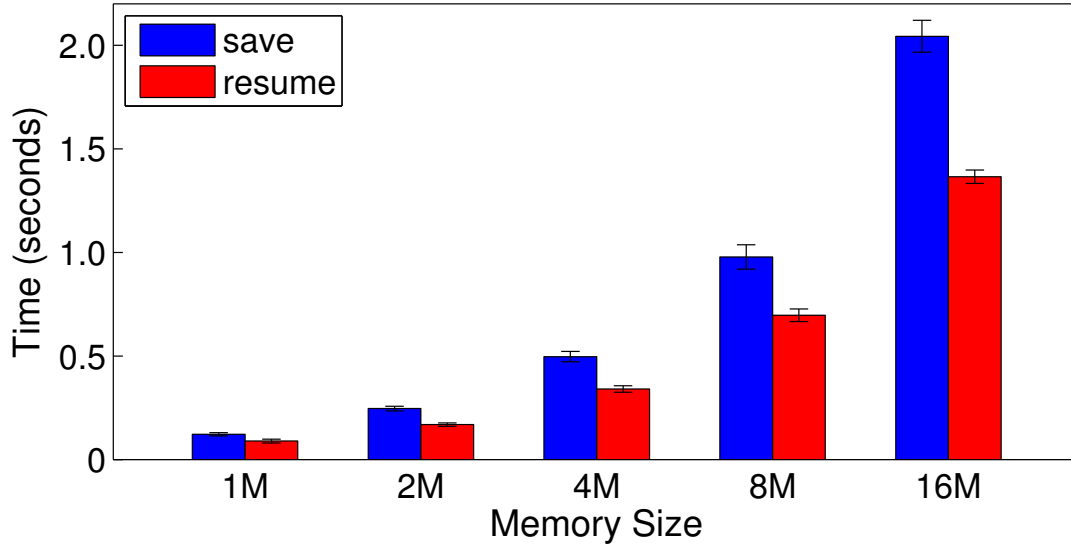
Here we measure the boot time of the MiniOS, which is defined as the time span from the time when Dom0 begins to create MiniOS to the time when the application is ready to run. The challenge is that we cannot directly use the timer/counter registers to record time information as we do in a non-virtualized environment, because the current Xen on ARM does not provide the wallclock time field. Therefore, we have to design our own approach to obtain two timestamps. The first timestamp is obtained when Dom0 issues the command for MiniOS creation. For the second timestamp, we modify MiniOS to invoke a hypercall to notify Xen hypervisor once the MiniOS finishes booting. Once receiving the hypercall, Xen reads its time register to get the second timestamp. Then, the boot time is the difference of the two timestamps. We have conducted 10 rounds of tests, and the results are shown in Table 4.2. The average boot time is 45.4 milliseconds, with a standard deviation of 3.1 milliseconds.

**Table 4.2:** Boot time of MiniOS in milliseconds.

Round	1	2	3	4	5	6	7	8	9	10
Time	41	40	47	46	50	45	46	45	49	45

### 4.6.3 Save/Resume Time

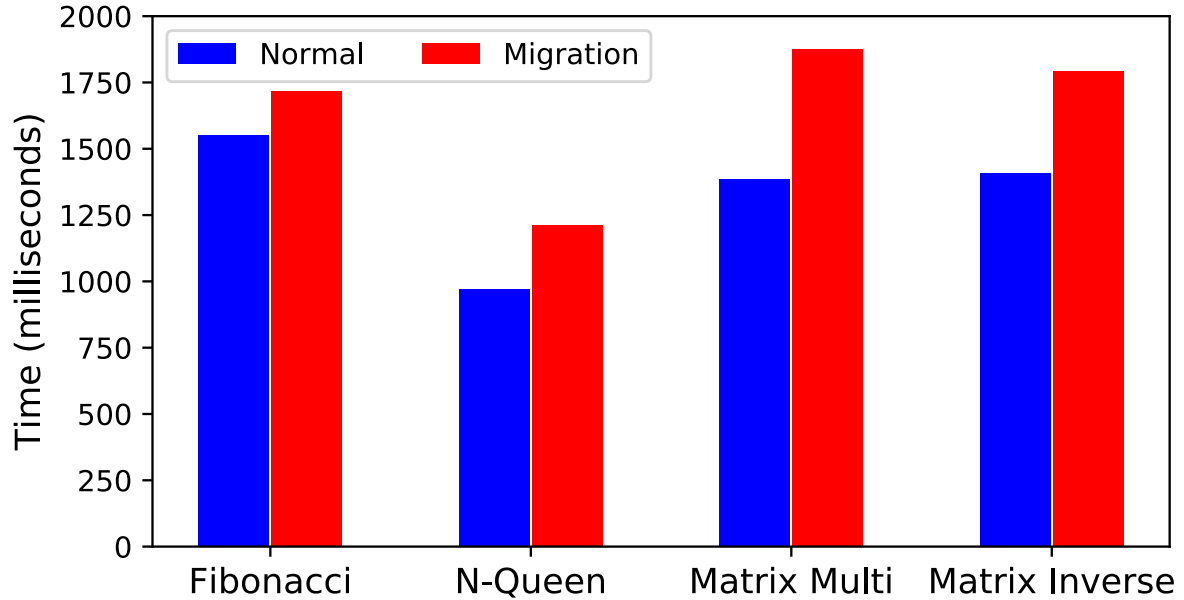
In this experiment, we investigate how fast the migration can be conducted. We focus on the *save time* and the *resume time* while ignoring the networking time from a smartphone to the cloud and that from the cloud to another smartphone. The save time is the time span from the time when the migration is triggered to the time once all the migration data, i.e. Delta, is stored into the disk (*Disk0* and *Disk1*). The resume time is the time span from the time when Dom0 of Xen receives all the migration data (i.e. Delta+Base) to the time when the (unfinished) application is ready to continue running. In our experiment, we choose a matrix multiplication task as the application, and deliberately choose different matrix size such that different size of memory has to be allocated for the migration data (Delta). Figure 4.6 shows the average of the save/resume time of 10 round tests along with error bar for different memory size. We can see that the save time is a little longer than the resume time, and both scale out almost linearly with regard to the memory size.



**Figure 4.6:** The save time and the resume time for different memory size.

#### 4.6.4 Migration overhead

In this evaluation, we measure and compare the execution time of four typical tasks under both normal and migration cases. These tasks are: (1) recursively calculating Fibonacci number ( $N = 30$ ), (2) finding all N-queens( $N = 9$ ) solutions, (3) multiplying two matrix( $500 \times 500$ ), and (4) calculating the inverse of a matrix( $500 \times 500$ ). Note that the first two tasks are computationally intensive while the last two are memory intensive. Figure 4.7 illustrates the barplot for the four tasks under normal and migration cases. Note that the migration time does not include the network transfer time since it depends on the network condition while this evaluation focuses on the migration overhead. From the results we can see that the computationally intensive tasks suffer less migration overhead than memory intensive tasks, since memory intensive tasks generate more Delta duration migration.

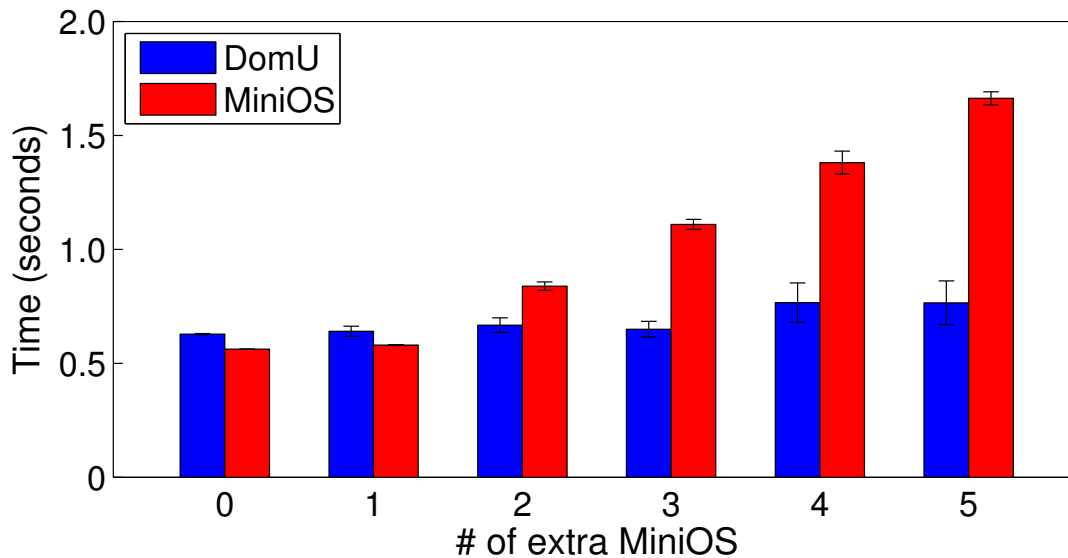


**Figure 4.7:** Execution time of tasks in normal and migration cases.

#### 4.6.5 Impact Among Domains

The smartphone in vRENT can run multiple MiniOSes concurrently, with each for a single application. One would wonder whether these MiniOSes affect owner's Android OS and whether they themselves affect one another. To answer these questions, we design the following experiment. We

let both android OS and the first MiniOS run a matrix multiplication task with each matrix of  $300 \times 300$ . We then create  $n$  extra MiniOSes to run concurrently,  $n \in [1, 5]$ , each with an identical CPU intensive task. Android OS has a CPU weight of 2048, and each MiniOS has a weight of  $256^2$ . We run the experiment for 10 rounds, and measure the execution time on Android OS and that on MiniOS for the same matrix multiplication task for each round. The average and the standard deviation of the execution time regarding different number of extra MiniOSes are shown in Figure 4.8. We can see that the impact of MiniOSes on Android OS is negligible, while MiniOSes do contend with each other. Therefore, the user experience of the smartphone owner is guaranteed, as long as a MiniOS hasn't claimed more weights to overpower other MiniOSes.



**Figure 4.8:** The execution time for different number of extra MiniOSes.

## 4.7 Discussion and Limitations

In this section, we will discuss limitations of our work and directions of future work.

---

<sup>2</sup>Each domain is assigned a weight, ranging from 1 to 65535. A domain with a weight of 512 will get twice as much CPU as a domain with a weight of 256 on a contended host.



### 4.7.1 Battery Drain and Wireless Charge

We admit that the battery drain will be the primary concern of smartphones serving as lightweight and pervasive edge nodes. However, such concern has been mitigated given the following factors:

- While no breakthrough on battery techniques, the manufactures are making smartphones with larger display sizes, smaller hardware components and therefore with more room for batteries [65]. Additionally, our solution uses smartphone only as data aggregation and processing hub without screen display that is the biggest battery hunger of smartphone.
- Our lightweight migration potentially allows users to select energy-efficient migration targets based on the energy level indicator. We may also allow user to specify her own preferences, such as the safe line of the battery level.
- The future adoption of long range wireless charge techniques [62, 85, 150] will make the battery drain problem trivial.

### 4.7.2 Incentives and Bootstrapping

We have not proposed an incentive mechanism or bootstrapping strategy for our Edge-IoT system, as we focus more on the technical side. Here, we want to give a discussion on the use case of incentive mechanism. However, we do need future work to test and verify those incentive mechanism and bootstrapping strategy. The key of the success of such system is the incentive mechanism, which makes the bootstrapping easier if such mechanism is popular to smartphone owners. Taking smart meter as a use case, utility company may adopt our solution by recruiting its clients directly by offering lower utility rate. It is a win-win strategy as the utility company can cut off relevant budgets of smart meter hub hardware, deployment and maintenance while the clients who join the program can enjoy lower utility rates as incentives. Similar use cases can be found in bring-your-own-device scenarios where peripherals will use smartphones as pervasive edge nodes and owners will get reimbursements for their usages.

## 4.8 Chapter Summary

In this chapter, we present vRENT, a novel methodology that allows smartphone owners to rent their idle resources to third party for managing IoT devices. vRENT can guarantee the security of both smartphone users and third party renters by leveraging Xen for ARM and MiniOS. Specifically, smartphone owners run their Android OS as a guest OS and rent their idle resources in the form of MiniOS. Meanwhile, in order to handle the situation that smartphone users walk out of the range of IoT devices while the tasks in MiniOS have not been finished, vRENT provides a light-weight migration method that allows MiniOS to migrate the unfinished tasks to the cloud or other smartphones nearby at a very low cost. We have implemented a prototype of vRENT on development boards and conducted extensive evaluations. Our experimental results show that vRENT is practical and the migration method is effective and efficient.

## Chapter 5

# Conclusion and Future Work

This dissertation explores the feasibility of applying edge computing on mobile and IoT devices to enhance their security and usability within a homogeneous environment. In particular, we first present a novel software-based trusted execution environment (TEE) for mobile applications based on an edge node and a hypervisor. This TEE ensures no exposure of data in memory, storage, and I/O by sending the mobile app computation and storage to the remote server and securing the I/O channel via encryption. Then, we present a comprehensive, flexible and efficient mobile app testing tool based on Edge computing, which is able to accurately record and replay all mobile input data in real time, including all touchscreen gestures and all sensor data, and outperforms the state of the art. Finally, we consider the challenges and opportunities that arise from Edge computing in IoT and present a new methodology that allows smartphone owners to rent their idle resource to third parties and migrates running apps when necessary. The three major contributions of this dissertation are summarized as follows.

- While mobile devices are getting popular and increasingly used to access sensitive data, we summary the security and management challenges that the mobile device could bring. We systematically analyze the possible attacks that could be conducted by a malicious attacker after she gains the root privilege of the system. In order to prevent these potential attacks, we design and implement vTrust, a system combines the hypervisor and Edge computing techniques and could prevent any data leakage even in the situation that the attacker has full control of the mobile Operating System. We conduct an extensive evaluation and prove that

although vTrust brings overhead on virtualization and data transmission, it is practical and effective in protecting the sensitive data from being leaked out.

- The ability to record and replay the operations on a smartphone app is useful in many contexts: recording a user’s interaction and replaying it for profiling and measuring, reproducing bugs to support debugging, generating inputs to support dynamic analysis and testing. While useful, this task has been proven difficult: a smartphone usually has a lot of input data, such as touch screen data, sensor data, while Android system does not provide enough APIs for record and replay tools to collect these data without root privilege for security concerns. We present MobiPlay, a novel record and replay tools that leverage edge computing to host the to-be-test app on an edge node, and use a user-level proxy app to send/receive the input/output data to/from the edge node. This method allows us to capture all the input data of the smartphone without modification. Experimental results show that MobiPlay successfully records and replay a selection of tens of most popular apps from Google Play with precise timing.
- The Internet of Things (IoT) is generating an unprecedented volume of data that is increasingly challenging to process. One promising solution is to backup IoT with Edge computing. However, the deployment of edge nodes is far from enough. To mitigate this problem, we present a solution that allows smartphone owner to rent their idle resource to the Third-Party for IoT management. In order to secure the data of both the renter and the smartphone owner, we design and implement vRent, a system that allows mobile device owners to rent their spare resource in term of MiniOS. Moreover, vRent develop an efficient migration strategy that enables the MiniOS to migrate to another smartphone when needed. Experimental results show that our solution is effective and incurs trivial performance overhead.

The experience we have learned from these three projects is that, in order to apply Edge computing to a new field, we need to provide a homogeneous environment for the upper layer apps. Thus, we need the help of virtualization techniques such as Hypervior or Container, which manages the hardware while provides an identical virtualized environment for the upper layer operating system and applications. For edge nodes which are usually deployed with x86-based machines, we have many mature virtualization technologies, such as VMWare, Xen, and KVM. For the mobile devices

which are usually based on the ARM architecture, we currently only have two virtualization options, KVM for ARM and Xen for ARM. Both solutions have their strength and weakness. For example, KVM for ARM (Adopted by vTrust) is easier to port and program on it while Xen (Adopted by vRent) is more efficient and has its own MiniOS.

In our future work, we will keep focusing on applying Edge computing to new areas and improving systems we designed. Specifically, we will work on the following aspects.

- **Compression Overhead.** vRent and MobiPlay still have 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 maintaining a high compression ratio, we can leverage hardware-assisted lossless compression techniques, such as H.265 [12] and VP9 [16]. These techniques are very efficient, we believe vTrust and MobiPlay could have a much shorter latency and higher FPS with them.
- **Advanced protection.** We have mentioned that vTrust can be built on a 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 by themselves. 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 [61], in our remote VMs.
- **Input data for different mobile devices** MobiPlay should be able to replay an app on a mobile device with the input data recorded from another device with distinct device configurations, such as resolution, screen size, etc. Therefore, one direction is to test and evaluate the cross-device portability on MobiPlay.
- **Support more input / output devices** Currently, many I/O devices are not supported by vRent, MobiPlay, and vTrust, such as camera and microphone. We believe it is worth improving these systems such that they can support all these devices.

# Bibliography

- [1] Encrypting file system in windows xp and windows server 2003. <https://technet.microsoft.com/en-us/enus/library/bb457065.aspx>, 2003.
- [2] Digital government - bring your own device. <https://obamawhitehouse.archives.gov/digitalgov/bring-your-own-device>, August 2012.
- [3] 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.
- [4] U.s. military phones: Android is system of choice. <http://www.zdnet.com/article/us-military-phones-android-is-system-of-choice/>, February 2012.
- [5] Advanced persistent threats now hitting mobile devices. <https://www.networkworld.com/article/2173639/wireless/advanced-persistent-threats-now-hitting-mobile-devices.html>, December 2013.
- [6] Serial accelerometer dongle. <https://www.sparkfun.com/products/retired/10537>, November 2014.
- [7] The two-phone solution doubles the pain, not the pleasure. [https://www.huffingtonpost.com/rebecca-abrahams/the-two-phone-solution-do\\_b\\_5085527.html](https://www.huffingtonpost.com/rebecca-abrahams/the-two-phone-solution-do_b_5085527.html), 2014.
- [8] Virtual smart phones in the cloud. <https://svmp.github.io/>, November 2014.
- [9] Android hardware abstraction layer documentation. <https://source.android.com/devices/halref/index.html>, February 2015.
- [10] Arm trustzone technology for cortex-a and cortex-m class processors. <http://www.arm.com/products/processors/technologies/trustzone/>, November 2015.
- [11] Mobile marketing statistics compilation. <http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/>, Jul 2015.
- [12] High efficiency video coding. [https://en.wikipedia.org/wiki/High\\_Efficiency\\_Video\\_Coding](https://en.wikipedia.org/wiki/High_Efficiency_Video_Coding), Jul 2016.
- [13] Lz4. <https://lz4.github.io/lz4/>, Jul 2016.
- [14] Raspberry pi touch display. <https://www.raspberrypi.org/products/raspberry-pi-touch-display/>, November 2016.

- [15] Virtual network computing. [https://en.wikipedia.org/wiki/Virtual\\_Network\\_Computing](https://en.wikipedia.org/wiki/Virtual_Network_Computing), 2016.
- [16] Vp9. <https://en.wikipedia.org/wiki/VP9>, Jul 2016.
- [17] Why carrying two phones has become a social embarrassment. <https://www.itproportal.com/2016/03/09/why-carrying-two-phones-has-become-a-social-embarrassment>, 2016.
- [18] 1mobile best android market. <http://www.1mobile.com/search.php?keywords=x86&submit=search>, May 2017.
- [19] Hackers are using android malware to spy on israeli military personnel. <https://thehackernews.com/2017/02/android-malware-israeli-military.html>, February 2017.
- [20] How Xen suspend and resume works. <https://mirage.io/wiki/xen-suspend>, 2017.
- [21] Ibm mobile solutions drive digital innovation. <http://www.ibm.com/mobile>, September 2017.
- [22] IntelÂ software guard extensions. <https://software.intel.com/en-us/sgx>, Jan 2017.
- [23] Linaro. <https://www.linaro.org>, 2017.
- [24] MiniOS on cubieboard2. <https://mirage.io/wiki/xen-on-cubieboard2>, 2017.
- [25] MiniOS on cubieboard2. <http://savannah.nongnu.org/projects/lwip/>, 2017.
- [26] 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](https://www.kaspersky.com/about/press-releases/2017_mobile-advertising-trojans), January 2017.
- [27] One kind of android smartphone ransomware is behind a massive rise in malicious software. <http://www.zdnet.com/article/>, June 2017.
- [28] What is regulatory compliance and why is it important? <https://www.thrivenetworks.com/blog/2017/07/27>, July 2017.
- [29] ABBOT JAVA GUI TEST FRAMEWORK. <http://abbot.sourceforge.net/doc/overview.shtml>.
- [30] DOMENICO AMALFITANO, ANNA RITA FASOLINO, AND PORFIRIO TRAMONTANA. A gui crawling-based technique for android mobile application testing. In *Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 252–261. IEEE, 2011.
- [31] DOMENICO AMALFITANO, ANNA RITA FASOLINO, PORFIRIO TRAMONTANA, BRYAN TA, AND ATIF MEMON. Mobiguitar—a tool for automated model-based testing of mobile apps. 2014.
- [32] AMAZON WEB SERVICE. Aws iot. <https://aws.amazon.com/iot/>, 2017.
- [33] ARDALAN AMIRI SANI, KEVIN BOOS, MIN HONG YUN, AND LIN 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, MobiSys '14*, pages 259–272, New York, NY, USA, 2014. ACM.

- [34] ANDROID MONKEY. <http://developer.android.com/tools/help/monkey.html>.
- [35] AHMED M AZAB, PENG NING, JITESH SHAH, QUAN CHEN, ROHAN BHUTKAR, GURUPRASAD GANESH, JIA MA, AND WENBO 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*, pages 90–102. ACM, 2014.
- [36] PAUL BARHAM, BORIS DRAGOVIC, KEIR FRASER, STEVEN HAND, TIM HARRIS, ALEX HO, ROLF NEUGEBAUER, IAN PRATT, AND ANDREW WARFIELD. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.
- [37] ANDREW BAUMANN, MARCUS PEINADO, AND GALEN HUNT. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.
- [38] JOHN BLACK AND PHILLIP ROGAWAY. Ciphers with arbitrary finite domains. In *Cryptographers’ Track at the RSA Conference*, pages 114–130. Springer, 2002.
- [39] FLAVIO BONOMI, RODOLFO MILITO, JIANG ZHU, AND SATEESH ADDEPALLI. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.
- [40] ROBERT BRADFORD, EVANGELOS KOTSOVINOS, ANJA FELDMANN, AND HARALD SCHIÖBERG. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 169–179. ACM, 2007.
- [41] DAVID BREITGAND, GILAD KUTIEL, AND DANNY RAZ. Cost-aware live migration of services in the cloud. In *SYSTOR*, 2010.
- [42] STEFAN BRENNER, COLIN WULF, DAVID GOLTZSCHE, NICO WEICHBRODT, MATTHIAS LORENZ, CHRISTOF FETZER, PETER R PIETZUCH, AND RÜDIGER KAPITZA. Securekeeper: Confidential zookeeper using intel sgx. In *Middleware*, page 14, 2016.
- [43] EDOUARD BUGNION, JASON NIEH, AND DAN TSAFRIR. Hardware and software support for virtualization. *Synthesis Lectures on Computer Architecture*, 12(1):1–206, 2017.
- [44] BRIAN BURG, RICHARD BAILEY, ANDREW J KO, AND MICHAEL D ERNST. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 473–484. ACM, 2013.
- [45] LIANG CAI AND HAO CHEN. Touchlogger: Inferring keystrokes on touch screen from smart-phone motion. *HotSec*, 11:9–9, 2011.
- [46] BJÖRN CASSENS, SIMON RIPPERGER, MARTIN HIEROLD, FRIEDER MAYER, AND RÜDIGER KAPITZA. Automated encounter detection for animal-borne sensor nodes. Uppsala, Sweden, 2017.
- [47] HAIBO CHEN, FENGZHE ZHANG, CHENG CHEN, ZIYE YANG, RONG CHEN, BINYU ZANG, AND WENBO MAO. Tamper-resistant execution in an untrusted operating system using a virtual machine monitor. 2007.



- [48] XIAOXIN CHEN, TAL GARFINKEL, E. CHRISTOPHER LEWIS, PRATAP SUBRAHMANYAM, CARL A. WALDSPURGER, DAN BONEH, JEFFREY DWOSKIN, AND DAN R.K. PORTS. Over-shadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS '08*, 2008.
- [49] YUE CHEN, YULONG ZHANG, ZHI WANG, AND TAO WEI. Downgrade attack on trustzone. *arXiv preprint arXiv:1707.05082*, 2017.
- [50] YUEQIANG CHENG AND XUHUA DING. Virtualization based password protection against malware in untrusted operating systems. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12, pages 201–218, Berlin, Heidelberg, 2012. Springer-Verlag.
- [51] YUEQIANG CHENG, XUHUA DING, AND R DENG. Appshield: Protecting applications against untrusted operating system. *Singapore Management University Technical Report, SMU-SIS-13*, 101, 2013.
- [52] YEONGPIL CHO, JUN-BUM SHIN, DONGHYUN KWON, MYUNGJOO HAM, YUNA KIM, AND YUNHEUNG PAEK. Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices. In *USENIX Annual Technical Conference*, pages 565–578, 2016.
- [53] BYUNG-GON CHUN, SUNGHWAN IHM, PETROS MANIATIS, MAYUR NAIK, AND ASHWIN PATTI. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.
- [54] CHRISTOPHER CLARK, KEIR FRASER, STEVEN HAND, JACOB GORM HANSEN, ERIC JUL, CHRISTIAN LIMPACH, IAN PRATT, AND ANDREW WARFIELD. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.
- [55] KEVIN M CONROY, MARK GRECHANIK, MATTHEW HELDIGE, EDY S LIONGOSARI, AND QING XIE. Automatic test generation from gui applications for testing web services. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 345–354. IEEE, 2007.
- [56] EDUARDO CUERVO, ARUNA BALASUBRAMANIAN, DAE-KI CHO, ALEC WOLMAN, STEFAN SAROIU, RANVEER CHANDRA, AND PARAMVIR BAHL. Maui: making smartphones last longer with code offload. In *MobiSys '10*.
- [57] SOPHIE CURTIS. Quarter of the world will be using smartphones in 2016. <http://www.telegraph.co.uk/technology/mobile-phones/11287659/Quarter-of-the-world-will-be-using-smartphones-in-2016.html>, Dec 2014.
- [58] CHRISTOFFER DALL AND JASON NIEH. Kvm/arm: the design and implementation of the linux arm hypervisor. In *ACM SIGPLAN Notices*, volume 49, pages 333–348. ACM, 2014.
- [59] SARAH M DIESBURG AND AN-I ANDY WANG. A survey of confidential data storage and deletion methods. *ACM Computing Surveys (CSUR)*, 43(1):2, 2010.

- [60] GEORGE W DUNLAP, SAMUEL T KING, SUKRU CINAR, MURTAZA A BASRAI, AND PETER M CHEN. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.
- [61] WILLIAM ENCK, PETER GILBERT, SEUNGYEOP HAN, VASANT TENDULKAR, BYUNG-GON CHUN, LANDON P COX, JAEYEON JUNG, PATRICK MCDANIEL, AND ANMOL N SHETH. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smart-phones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.
- [62] ENERGIOUS. Wire-Free Charging Technology. <http://energious.com/>, 2017.
- [63] ESPRESSO. <https://developer.android.com/tools/testing-support-library/index.html>.
- [64] FITBIT. Fitness tracker. <https://www.fitbit.com/home>, 2017.
- [65] FORTUNE. Apple could bundle a bigger battery into a smaller iphone. [urlhttp://fortune.com/2017/02/14/apple-iphone-8-battery/](http://fortune.com/2017/02/14/apple-iphone-8-battery/), 2017.
- [66] ROXANA GEAMBASU, JOHN P JOHN, STEVEN D GRIBBLE, TADAYOSHI KOHNO, AND HENRY M LEVY. Keypad: an auditing file system for theft-prone devices. In *Proceedings of the sixth conference on Computer systems*, pages 1–16. ACM, 2011.
- [67] LORENZO GOMEZ, IULIAN NEAMTIU, TAYYABA AZIM, AND TODD MILLSTEIN. Reran: Timing-and touch-sensitive record and replay for android. In *35th International Conference on Software Engineering (ICSE)*, pages 72–81. IEEE, 2013.
- [68] GOOGLE. Google cloud internet of things solution. <https://cloud.google.com/solutions/iot/>, 2017.
- [69] GOOGLE INC. Android thing. <https://developer.android.com/things/index.html>, 2017.
- [70] MARK S GORDON, D ANOUSHE JAMSHIDI, SCOTT MAHLKE, Z MORLEY MAO, AND XU CHEN. Comet: code offload by migrating execution transparently. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 93–106, 2012.
- [71] MARK GRECHANIK, QING XIE, AND CHEN FU. Creating gui testing tools using accessibility technologies. In *International Conference on Software Testing, Verification and Validation Workshops*, pages 243–250. IEEE, 2009.
- [72] LE GUAN, PENG LIU, XINYU XING, XINYANG GE, SHENGZHI ZHANG, MENG YU, AND TRENT JAEGER. Trustshadow: Secure execution of unmodified applications with arm trust-zone. *arXiv preprint arXiv:1704.05600*, 2017.
- [73] GUITAR. <http://sourceforge.net/p/guitar/wiki/Home/>.
- [74] TRINABH GUPTA, RAYMAN PREET SINGH, AMAR PHANISHAYEE, JAEYEON JUNG, AND RATUL MAHAJAN. Bolt: Data management for connected homes. In *NSDI*, pages 243–256, 2014.

- [75] MATTHEW HALPERN, YUHAO ZHU, RAMESH PERI, AND VIJAY JANAPA REDDI. Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 215–224. IEEE, 2015.
- [76] HAO HAN, SHANHE YI, QUN LI, GUOBIN SHEN, AND ED NOVAK. Amil: Localizing neighboring mobile devices through a simple gesture. In *INFOCOM, 2016 Proceedings IEEE*, April 2016.
- [77] ZIJIANG HAO, ED NOVAK, SHANHE YI, AND QUN LI. Challenges and software architecture for fog computing. *IEEE Internet Computing*, 21(2):44–53, 2017.
- [78] WOLFGANG HEIDER, RICK RABISER, AND PAUL GRÜNBACHER. Facilitating the evolution of products in product line engineering by capturing and replaying configuration decisions. *International Journal on Software Tools for Technology Transfer*, 14(5):613–630, 2012.
- [79] OWEN S. HOFMANN, SANGMAN KIM, ALAN M. DUNN, MICHAEL Z. LEE, AND EMMETT WITCHEL. Inktag: Secure applications on an untrusted operating system. In *ASPLOS '13*, 2013.
- [80] KIRAK HONG, DAVID LILLETHUN, UMAKISHORE RAMACHANDRAN, BEATE OTTENWÄLDER, AND BORIS KOLDEHOFE. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*, pages 15–20. ACM, 2013.
- [81] CHIH-WEI HUANG. Android-x86 project. <http://www.android-x86.org/>, November 2014.
- [82] JIAN HUANG, ANIRUDH BADAM, RANVEER CHANDRA, AND EDMUND B NIGHTINGALE. Weardrive: Fast and energy-efficient storage for wearables. In *USENIX Annual Technical Conference*, pages 613–625, 2015.
- [83] IBM. Waston internet of things. <https://www.ibm.com/internet-of-things/>, 2017.
- [84] JACARETO. <http://sourceforge.net/projects/jacareto/>.
- [85] JOUYA JADIDIAN AND DINA KATABI. Magnetic mimo: How to charge your phone in your pocket. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 495–506. ACM, 2014.
- [86] JFCUNIT. <http://jfcunit.sourceforge.net/>.
- [87] HAI JIN, LI DENG, SONG WU, XUANHUA SHI, AND XIAODONG PAN. Live virtual machine migration with adaptive, memory compression. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.
- [88] MILAN JOVIC, ANDREA ADAMOLI, DMITRIJS ZAPARANUKS, AND MATTHIAS HAUSWIRTH. Automating performance testing of interactive java applications. In *Proceedings of the 5th Workshop on Automation of Software Test*, pages 8–15. ACM, 2010.
- [89] ROELOF KEMP, NICHOLAS PALMER, THILO KIELMANN, AND HENRI E BAL. Cuckoo: A computation offloading framework for smartphones. In *MobiCASE*, pages 59–79. Springer, 2010.

- [90] SHAHEDUL HUQ KHANDKAR, SM SOHAN, JONATHAN SILLITO, AND FRANK MAURER. Tool support for testing complex multi-touch gestures. In *International Conference on Interactive Tabletops and Surfaces*, pages 59–68. ACM, 2010.
- [91] AVI KIVITY, YANIV KAMAY, DOR LAOR, URI LUBLIN, AND ANTHONY LIGUORI. Kvm: the linux virtual machine monitor. *Proceedings of the Linux Symposium*, 1:225–230, 2007.
- [92] DMITRII KUNAVSKII, OLEKSI OLEKSENKO, SERGEI ARNAUTOV, BOHDAN TRACH, PRAMOD BHATOTIA, PASCAL FELBER, AND CHRISTOF FETZER. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 205–221. ACM, 2017.
- [93] HEATHER 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.
- [94] CHENG LI, ZHENGRUI QIN, ED NOVAK, AND QUN LI. Securing sdn infrastructure of iot-fog network from mitm attacks. *IEEE Internet of Things Journal*, 2017.
- [95] TIANXING LI, CHUANKAI AN, XINRAN XIAO, ANDREW T. CAMPBELL, AND XIA ZHOU. Real-time screen-camera communication behind any scene. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 197–211, 2015.
- [96] YANLIN LI, JONATHAN M MCCUNE, JAMES NEWSOME, ADRIAN PERRIG, BRANDON BAKER, AND WILL DREWRY. Minibox: A two-way sandbox for x86 native code. In *USENIX Annual Technical Conference*, pages 409–420, 2014.
- [97] YUE LI, HAINING WANG, AND KUN SUN. A study of personal information in human-chosen passwords and its security implications. In *INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE*, pages 1–9. IEEE, 2016.
- [98] YUE LI, HAINING WANG, AND KUN SUN. Personal information in passwords and its security implications. *IEEE Transactions on Information Forensics and Security*, 6013:1–1, 2017.
- [99] ZHENJIANG LI, MO LI, JILIANG WANG, AND ZHICHAO CAO. Ubiquitous data collection for mobile users in wireless sensor networks. In *INFOCOM, 2011 Proceedings IEEE*, pages 2246–2254. IEEE, 2011.
- [100] STEFFEN LIEBERGELD AND MATTHIAS LANGE. Android security, pitfalls and lessons learned. In *Information Sciences and Systems 2013*, pages 409–417. Springer, 2013.
- [101] CHIA-CHI LIN, HONGYANG LI, XIAOYONG ZHOU, AND X WANG. Screenmilk: How to milk your android screen for secrets. In *21st Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA*, 2014.
- [102] PENG LIU, DALE WILLIS, AND SUMAN BANERJEE. Paradrop: Enabling lightweight multi-tenancy at the network’s extreme edge. In *Edge Computing (SEC), IEEE/ACM Symposium on*, pages 1–13. IEEE, 2016.

- [103] ANIL MADHAVAPEDDY, THOMAS LEONARD, MAGNUS SKJEGSTAD, THOMAS GAZAGNAIRE, DAVID SHEETS, DAVID J SCOTT, RICHARD MORTIER, AMIR CHAUDHRY, BALRAJ SINGH, JON LUDLAM, ET AL. Jitsu: Just-in-time summoning of unikernels. In *NSDI*, pages 559–573, 2015.
- [104] ANIL MADHAVAPEDDY, RICHARD MORTIER, CHARALAMPOS ROTSOS, DAVID SCOTT, BALRAJ SINGH, THOMAS GAZAGNAIRE, STEVEN SMITH, STEVEN HAND, AND JON CROWCROFT. Unikernels: Library operating systems for the cloud. In *ACM SIGPLAN Notices*, volume 48, pages 461–472. ACM, 2013.
- [105] MARATHONITE. <http://marathontesting.com/>.
- [106] L. MARIANI, M. PEZZE, O. RIGANELLI, AND M. SANTORO. Autoblacktest: Automatic black-box testing of interactive applications. In *proceedings of the Fifth International Conference on Software Testing, Verification and Validation (ICST)*, 2012.
- [107] JOAO MARTINS, MOHAMED AHMED, COSTIN RAICIU, VLADIMIR OLTEANU, MICHIO HONDA, ROBERTO BIFULCO, AND FELIPE HUICI. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473. USENIX Association, 2014.
- [108] JONATHAN M McCUNE, YANLIN LI, NING QU, ZONGWEI ZHOU, ANUPAM DATTA, VIRGIL GLIGOR, AND ADRIAN PERRIG. Trustvisor: Efficient tcb reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 143–158. IEEE, 2010.
- [109] JONATHAN M McCUNE, BRYAN J PARNO, ADRIAN PERRIG, MICHAEL K REITER, AND HIROSHI ISOZAKI. Flicker: An execution infrastructure for tcb minimization. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 315–328. ACM, 2008.
- [110] ROSS McILROY AND JOSEPH S SVENTEK. Hera-jvm: Abstracting processor heterogeneity behind a virtual machine. In *HotOS*, 2009.
- [111] YAN MICHALEVSKY, DAN BONEH, AND GABI NAKIBLY. Gyrophone: Recognizing speech from gyroscope signals. In *USENIX Security*, pages 1053–1067, 2014.
- [112] MICROSFOT. Azure iot. <https://www.microsoft.com/en-us/internet-of-things/azure-iot-suite>, 2017.
- [113] EMILIANO MILUZZO, ALEXANDER VARSHAVSKY, SUHRID BALAKRISHNAN, AND ROMIT ROY CHOUDHURY. Tapprints: your finger taps have fingerprints. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 323–336. ACm, 2012.
- [114] SUBHAS C MISRA AND VIRENDRA C BHAVSAR. Relationships between selected software measures and latent bug-density: Guidelines for improving quality. In *International Conference on Computational Science and Its Applications*, pages 724–732. Springer, 2003.
- [115] MONKEYRUNNER. <http://developer.android.com/tools/help/monkeyrunner-concepts.html>.

- [116] SATISH NARAYANASAMY, GILLES POKAM, AND BRAD CALDER. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 284–295. IEEE Computer Society, 2005.
- [117] E. NOVAK AND QUN LI. Near-pri: Private, proximity based location sharing. In *INFOCOM, 2014 Proceedings IEEE*, pages 37–45, April 2014.
- [118] ED NOVAK, YUTAO TANG, ZIJIANG HAO, QUN LI, AND YIFAN ZHANG. Physical media covert channels on smart mobile devices. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 367–378. ACM, 2015.
- [119] ED NOVAK, YUTAO TANG, ZIJIANG HAO, QUN LI, AND YIFAN ZHANG. Physical media covert channels on smart mobile devices. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp ’15, pages 367–378, New York, NY, USA, 2015. ACM.
- [120] EMMANUEL OWUSU, JUN HAN, SAUVIK DAS, ADRIAN PERRIG, AND JOY ZHANG. Accessory: password inference using accelerometers on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, page 9. ACM, 2012.
- [121] UNKYU PARK AND JOHN HEIDEMANN. Data muling with mobile phones for sensor networks. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, pages 162–175. ACM, 2011.
- [122] PCWORLD. Apple watch leans on the iphone for app management and heavy lifting. <http://www.pcworld.com/article/2689362/apple-watch-leans-on-the-iphone-for-app-management-and-heavy-lifting.html>, 2014.
- [123] POUNDER. <http://pounder.sourceforge.net/>.
- [124] ZHENGRUI QIN, YUTAO TANG, ED NOVAK, AND QUN LI. Mobisplay: A remote execution based record-and-replay tool for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering*, pages 571–582. ACM, 2016.
- [125] RATIONAL ROBOT. <http://www.ibm.com>.
- [126] ROBOTIUM. <https://code.google.com/p/robotium/>.
- [127] RUSTY RUSSELL. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.
- [128] SAMSUNG. Arndale board. [http://www.arndaleboard.org/wiki/index.php/Main\\_Page](http://www.arndaleboard.org/wiki/index.php/Main_Page), November 2014.
- [129] NUNO SANTOS, HIMANSHU RAJ, STEFAN SAROIU, AND ALEC WOLMAN. Trusted language runtime (tlr): enabling trusted applications on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pages 21–26. ACM, 2011.
- [130] NUNO SANTOS, HIMANSHU RAJ, STEFAN SAROIU, AND ALEC WOLMAN. Using arm trust-zone to build a trusted language runtime for mobile applications. In *ASPLOS ’14*, 2014.

- [131] CONSTANTINE P SAPUNTZAKIS, RAMESH CHANDRA, BEN PFAFF, JIM CHOW, MONICA S LAM, AND MENDEL ROSENBLUM. Optimizing the migration of virtual computers. *ACM SIGOPS Operating Systems Review*, 36(SI):377–390, 2002.
- [132] MAHADEV SATYANARAYANAN. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.
- [133] MAHADEV SATYANARAYANAN, PARAMVIR BAHL, RAMÓN CACERES, AND NIGEL DAVIES. The case for vm-based cloudlets in mobile computing. *Pervasive Computing*, 8(4):14–23, 2009.
- [134] MAHADEV SATYANARAYANAN, ZHUO CHEN, KIRYONG HA, WENLU HU, WOLFGANG RICHTER, AND PADMANABHAN PILLAI. Cloudlets: at the leading edge of mobile-cloud convergence. In *Mobile Computing, Applications and Services (MobiCASE), 2014 6th International Conference on*, pages 1–9. IEEE, 2014.
- [135] MAHADEV SATYANARAYANAN, PIETER SIMOENS, YU XIAO, PADMANABHAN PILLAI, ZHUO CHEN, KIRYONG HA, WENLU HU, AND BRANDON AMOS. Edge analytics in the internet of things. *IEEE Pervasive Computing*, 14(2):24–31, 2015.
- [136] FELIX SCHUSTER, MANUEL COSTA, CÉDRIC FOURNET, CHRISTOS GKANTSIDIS, MARCUS PEINADO, GLORIA MAINAR-RUIZ, AND MARK RUSSINOVICH. Vc3: Trustworthy data analytics in the cloud using sgx. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 38–54. IEEE, 2015.
- [137] SELENDROID. <http://selendroid.io>.
- [138] FAHAD SHAON, MURAT KANTARCIOGLU, ZHIQIANG LIN, AND LATIFUR KHAN. Sgx-bigmatrix: A practical encrypted data analytic framework with trusted processors. 2017.
- [139] WEISONG SHI, JIE CAO, QUAN ZHANG, YOUHUIZI LI, AND LANYU XU. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.
- [140] SUDARSHAN M SRINIVASAN, SRIKANTH KANDULA, CHRISTOPHER R ANDREWS, AND YUANYUAN ZHOU. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference, General Track*, pages 29–44. Boston, MA, USA, 2004.
- [141] NIST-FIPS STANDARD. Announcing the advanced encryption standard (aes). *Federal Information Processing Standards Publication*, 197:1–51, 2001.
- [142] HE SUN, KUN SUN, YUEWU WANG, JIWU JING, AND HAINING WANG. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*, pages 367–378. IEEE, 2015.
- [143] VIRTUAL OPEN SYSTEMS. Kvm on arm performance. <http://www.virtualopensystems.com/en/products/kvm-performance/>, November 2014.
- [144] VIRTUAL OPEN SYSTEMS. Kvm virtualization on arndale development board. <http://www.virtualopensystems.com/en/solutions/guides/kvm-virtualization-on-arndale/>, November 2014.

- [145] YANG TANG, PHILLIP AMES, SRAVAN BHAMIDIPATI, ASHISH BIJLANI, ROXANA GEAM-BASU, AND NIKHIL SARDA. Cleanos: Limiting mobile data exposure with idle eviction. In *OSDI*, volume 12, pages 77–91, 2012.
- [146] UI AUTOMATOR. <https://developer.android.com/tools/testing-support-library/index.html>.
- [147] DING WANG, ZIJIAN ZHANG, PING WANG, JEFF YAN, AND XINYI HUANG. Targeted online password guessing: An underestimated threat. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1242–1254. ACM, 2016.
- [148] NICO WEICHBRODT, ANIL KURMUS, PETER PIETZUCH, AND RÜDIGER KAPITZA. Async-shock: Exploiting synchronisation bugs in intel sgx enclaves. In *Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS 2016)*, 2016.
- [149] MARTIN WHITE, MARIO LINARES-VÁSQUEZ, PETER JOHNSON, CARLOS BERNAL-CÁRDENAS, AND DENYS POSHYVANYK. Generating reproducible and replayable bug reports from android application crashes. In *23rd IEEE International Conference on Program Comprehension (ICPC)*, 2015.
- [150] WI-CHARGE. WI-CHARGE. <https://www.wi-charge.com/>, 2017.
- [151] DALE F WILLIS, ARKODEB DASGUPTA, AND SUMAN BANERJEE. Paradrop: a multi-tenant platform for dynamically installed third party services on home gateways. In *SIGCOMM workshop on Distributed cloud computing*, pages 43–44. ACM, 2014.
- [152] YUBIN XIA, YUTAO LIU, CHENG TAN, MINGYANG MA, HAIBING GUAN, BINYU ZANG, AND HAIBO CHEN. Tinman: Eliminating confidential mobile data exposure with security oriented offloading. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys '15*, pages 27:1–27:16, New York, NY, USA, 2015. ACM.
- [153] ZHI XU, KUN BAI, AND SENCUN 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*, pages 113–124. ACM, 2012.
- [154] JISOO YANG AND KANG G SHIN. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 71–80. ACM, 2008.
- [155] SHUSEN YANG, USMAN ADEEL, JULIE MCCANN, ET AL. Selfish mules: Social profit maximization in sparse sensor networks using rationally-selfish human relays. *Selected Areas in Communications, IEEE Journal on*, 31(6):1124–1134, 2013.
- [156] SHANHE YI, ZIJIAN HAO, ZHENGRUI QIN, AND QUN LI. Fog computing: Platform and applications. In *Hot Topics in Web Systems and Technologies (HotWeb), 2015 Third IEEE Workshop on*, pages 73–78. IEEE, 2015.
- [157] SHANHE YI, CHENG LI, AND QUN LI. A survey of fog computing: concepts, applications and issues. In *Proceedings of the 2015 Workshop on Mobile Big Data*, pages 37–42. ACM, 2015.



- [158] SHANHE YI, ZHENGRUI QIN, AND QUN LI. Security and privacy issues of fog computing: A survey. In *International Conference on Wireless Algorithms, Systems, and Applications*, pages 685–695. Springer, 2015.
- [159] SHANHE YI, ZHENGRUI QIN, ED NOVAK, YAFENG YIN, AND QUN LI. Glassgesture: Exploring head gesture interface of smart glasses. In *INFOCOM, 2016 Proceedings IEEE*, April 2016.
- [160] THOMAS ZACHARIAH, NOAH KLUGMAN, BRADFORD CAMPBELL, JOSHUA ADKINS, NEAL JACKSON, AND PRABAL DUTTA. The internet of things has a gateway problem. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, pages 27–32. ACM, 2015.
- [161] BEN ZHANG, NITESH MOR, JOHN KOLB, DOUGLAS S CHAN, KEN LUTZ, ERIC ALLMAN, JOHN WAWRZYNEK, EDWARD A LEE, AND JOHN KUBIATOWICZ. The cloud is not enough: Saving iot from the cloud. In *HotCloud*, 2015.
- [162] WENZHANG ZHU, CHO-LI WANG, AND FRANCIS CM LAU. Jessica2: A distributed java virtual machine with transparent thread migration support. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 381–388. IEEE, 2002.