DESIGNING EFFECTIVE SECURITY AND PRIVACY SCHEMES FOR WIRELESS MOBILE DEVICES

A Dissertation
Submitted to
the Temple University Graduate Board

in Partial Fulfillment
of the Requirements for the Degree of
DOCTOR OF PHILOSOPHY

by

Wu, Longfei
August, 2017

Examing Committee Members:

Dr. Xiaojiang Du, Dept. of Computer and Information Sciences (Advisor)
Dr. Jie Wu, Dept. of Computer and Information Sciences (Committee chair)
Dr. Bo Ji, Dept. of Computer and Information Sciences
Dr. Yimin Zhang, Dept. of Electrical and Computer Engineering (External reader)
ABSTRACT

Designing Effective Security and Privacy Schemes for Wireless Mobile Devices

by

Wu, Longfei

The growing ubiquity of modern wireless and mobile electronic devices has brought our daily lives with more convenience and fun. Today’s smartphones are equipped with a variety of sensors and wireless communication technologies, which can support not only the basic functions like phone call and web browsing, but also advanced functions like mobile pay, biometric security, fitness monitoring, etc. Internet-of-Things (IoT) is another category of popular wireless devices that are networked to collect and exchange data. For example, the smart appliances are increasingly deployed to serve in home and office environments, such as smart thermostat, smart bulb, and smart meter. Additionally, implantable medical devices (IMD) is the typical type of modern wireless devices that are implanted within human body for diagnostic, monitoring, and therapeutic purposes. However, these modern wireless and mobile devices are not well protected compared with traditional personal computers (PCs), due to the intrinsic limitations in computation power, battery capacity, etc. In this dissertation, we first present the security and privacy vulnerabilities we discovered. Then, we present our designs to address these issues and enhance the security of smartphones, IoT devices, and IMDs.

For smartphone security, we investigate the mobile phishing attacks, mobile clickjacking attacks and mobile camera-based attacks. Phishing attacks aim to steal private information such as credentials. We propose a novel anti-phishing scheme MobiFish, which can detect both phishing webpages and phishing applications (apps). The key idea is to check the consistency between the claimed identity and the actual identity of a webpage/app. The claimed identity can be extracted from the
screenshot of login user interface (UI) using the optical character recognition (OCR) technique, while the actual identity is indicated by the secondary-level domain name of the Uniform Resource Locator (URL) to which the credentials are submitted. Clickjacking attacks intend to hijack user inputs and re-route them to other UIs that are not supposed to receive them. To defend such attacks, a lightweight and independent detection service is integrated into the Android operating system. Our solution requires no user or app developer effort, and is compatible with existing commercial apps. Camera-based attacks on smartphone can secretly capture photos or videos without the phone user’s knowledge. One advanced attack we discovered records the user’s eye movements when entering passwords. We found that it is possible to recover simple passwords from the video containing user eye movements.

Next, we propose an out-of-band two-factor authentication scheme for indoor IoT devices (e.g., smart appliances) based on the Blockchain infrastructure. Since smart home environment consists of multiple IoT devices that may share their sensed data to better serve the user, when one IoT device is being accessed, our design utilizes another device to conduct a secondary authentication over an out-of-band channel (light, acoustic, etc.), to detect if the access requestor is a malicious external device.

Unlike smartphones and IoT devices, IMDs have the most limited computation and battery resources. We devise a novel smartphone-assisted access control scheme in which the patient’s smartphone is used to delegate the heavy computations for authentication and authorization. The communications between the smartphone and the IMD programmer are conducted through an audio cable, which can resist the wireless eavesdropping and other active attacks.

**Keywords:** Security, privacy, smartphone, phishing attacks, clickjacking attacks, camera-based attacks, Internet-of-Things, secondary authentication, out-of-band, implantable medical devices, access control, attribute-based encryption.
To my beloved parents
ACKNOWLEDGEMENTS

My doctoral study at Temple University has been and always will be one of the most valuable experience in my life. I have benefited a lot from the challenges and difficulties that I encountered and finally managed to overcome during the past five years. Through this process, I have established a rigorous attitude towards research and have developed the abilities required as a qualified researcher. I would like to express my sincere gratitude to everyone who has helped and supported me.

Firstly, I would like to extend my deep appreciation to my advisor Dr. Xiaojiang Du, who has been guiding me and encouraging me throughout the Ph.D. program. Without his innumerable and continuous support, I would have not been able to accomplish as much as I have. Dr. Du has devoted his time and efforts advising my research, selecting cutting-edge topics for me and sharing his insightful ideas with me. He has also taught me various methodologies like self-study and critical thinking, which I can use in my whole life.

I am also very grateful to the professors who serve in my dissertation examining committee. Dr. Jie Wu and Dr. Bo Ji have been advising my Ph.D. progress all the way from the preliminary exam 1, preliminary exam 2, to the dissertation defense. They were also the co-authors of my papers, and provided many precious comments and feedbacks. Additionally, I would like to thank Dr. Yimin Zhang, who spent his valuable time examining my dissertation defense as the external reader.

It was my great pleasure to meet all my colleagues and visiting scholars in the department. I appreciate their friendship and the happy time we spend together. My special thanks go to Dr. Jingwei Liu, Dr. Wei Chang, Dr. Xiali Hei, and Dr. Xueli Huang, who gave me tons of help and advices during the job interviews and
dissertation preparation.

Finally, I would like to express my deepest gratitude to my parents for their emotional and spiritual support. Their love and encouragement are the strongest driving force for the completion of my doctorate program.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>DEDICATION</td>
<td>v</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xiv</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Motivation and Challenge</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Major Contributions</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Dissertation Overview</td>
<td>8</td>
</tr>
<tr>
<td>2. MOBILE PHISHING ATTACKS</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Background</td>
<td>11</td>
</tr>
<tr>
<td>2.2.1 Hardware Limitations</td>
<td>12</td>
</tr>
<tr>
<td>2.2.2 User Habits and Preferences</td>
<td>14</td>
</tr>
<tr>
<td>2.3 Mobile Phishing Attack Models</td>
<td>14</td>
</tr>
<tr>
<td>2.3.1 Mobile Web Page Phishing Attacks</td>
<td>15</td>
</tr>
<tr>
<td>2.3.2 Mobile Application Phishing Attacks</td>
<td>17</td>
</tr>
<tr>
<td>2.3.3 Mobile Account Registry Phishing Attacks</td>
<td>17</td>
</tr>
<tr>
<td>2.4 Overview of MobiFish Scheme</td>
<td>22</td>
</tr>
<tr>
<td>2.4.1 Motivation</td>
<td>22</td>
</tr>
<tr>
<td>2.4.2 Identity Extraction</td>
<td>22</td>
</tr>
<tr>
<td>2.4.3 OCR Techniques</td>
<td>23</td>
</tr>
<tr>
<td>2.5 The MobiFish Scheme</td>
<td>25</td>
</tr>
<tr>
<td>2.5.1 The WebFish Scheme</td>
<td>25</td>
</tr>
<tr>
<td>2.5.2 The AppFish Scheme</td>
<td>27</td>
</tr>
<tr>
<td>2.5.3 The AccountFish Scheme</td>
<td>30</td>
</tr>
<tr>
<td>2.6 Implementation and Performance Evaluation</td>
<td>32</td>
</tr>
<tr>
<td>2.6.1 Experiments with WebFish</td>
<td>32</td>
</tr>
</tbody>
</table>
5. TWO FACTOR AUTHENTICATION SCHEME FOR INTERNET-OF-THINGS

5.1 Introduction
5.2 Adversary Model
5.3 The Out-of-band Two-factor Authentication Scheme
5.4 Implementation and Evaluation
   5.4.1 Evaluation Environment Setup
   5.4.2 Relationship Establishment on Blockchain
   5.4.3 Blockchain Performance
5.5 Literature Review
5.6 Chapter Summary

6. SMARTPHONE-BASED ACCESS CONTROL SCHEME FOR IMPLANTABLE MEDICAL DEVICES

6.1 Introduction
6.2 Motivation
   6.2.1 Special Use Cases
   6.2.2 Adversary Model
   6.2.3 Access Control Architecture
   6.2.4 Access Control Model
   6.2.5 Communications through Audio Cable
6.3 Attribute-based Encryption
6.4 Protocol Design
   6.4.1 System Overview
   6.4.2 Access Control Protocol Design
6.5 Security Analysis
   6.5.1 Resistance to Passive Attacks
   6.5.2 Resistance to Active Attacks
   6.5.3 Other Attacks
6.6 Evaluation
   6.6.1 Prototype Implementation
   6.6.2 Testbed Specification
   6.6.3 Experimental Results
6.7 Literature Review
6.8 Chapter Summary

7. DISSERTATION CONCLUSION

7.1 Summary of Contributions
7.2 Future Research
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Display of URL in Mobile Browser (Android)</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>URL Letter Replacement</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Comparison of Real and Fake Ebay Login Pages</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>Persistent Account Registry Function</td>
<td>18</td>
</tr>
<tr>
<td>2.5</td>
<td>Real and Fake Twitter Login User Interface</td>
<td>20</td>
</tr>
<tr>
<td>2.6</td>
<td>Comparison of OCR Performance on Mobile Phone and PC</td>
<td>24</td>
</tr>
<tr>
<td>2.7</td>
<td>The Work Flow of WebFish</td>
<td>26</td>
</tr>
<tr>
<td>2.8</td>
<td>The Work Flow of AppFish</td>
<td>28</td>
</tr>
<tr>
<td>2.9</td>
<td>The Work Flow of AccountFish</td>
<td>31</td>
</tr>
<tr>
<td>2.10</td>
<td>Experiments with WebFish on Mobile Phishing Login Page</td>
<td>33</td>
</tr>
<tr>
<td>2.11</td>
<td>Experiments with WebFish on Conventional Phishing Login Page</td>
<td>34</td>
</tr>
<tr>
<td>2.12</td>
<td>Experiments with AppFish</td>
<td>36</td>
</tr>
<tr>
<td>2.13</td>
<td>Experiments with AccountFish</td>
<td>37</td>
</tr>
<tr>
<td>3.1</td>
<td>Illustration of Clickjacking Attacks</td>
<td>48</td>
</tr>
<tr>
<td>3.2</td>
<td>Clickjacking Attacks on third-party Client Apps</td>
<td>63</td>
</tr>
<tr>
<td>3.3</td>
<td>CDS and the Four Probes</td>
<td>67</td>
</tr>
<tr>
<td>3.4</td>
<td>Customized Settings Menu</td>
<td>69</td>
</tr>
<tr>
<td>3.5</td>
<td>Clickjacking Attack and Defense</td>
<td>70</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Summary of Tested URLs</td>
<td>35</td>
</tr>
<tr>
<td>2.2</td>
<td>Average Execution Time of the MobiFish Scheme</td>
<td>38</td>
</tr>
<tr>
<td>3.1</td>
<td>Window Types for Floating Window</td>
<td>53</td>
</tr>
<tr>
<td>3.2</td>
<td>Window Flags for Floating Window</td>
<td>54</td>
</tr>
<tr>
<td>3.3</td>
<td>Clickjacking Attacks on System Apps</td>
<td>61</td>
</tr>
<tr>
<td>4.1</td>
<td>Passcode Inference Results for 4-digit Passcodes</td>
<td>91</td>
</tr>
<tr>
<td>5.1</td>
<td>Emulator Device Specification</td>
<td>103</td>
</tr>
<tr>
<td>5.2</td>
<td>Summary of CPU Usage</td>
<td>106</td>
</tr>
<tr>
<td>6.1</td>
<td>Parameters used for modulation and demodulation</td>
<td>133</td>
</tr>
</tbody>
</table>
Recent years have witnessed the rapidly growing popularity of wireless and mobile devices including, but not limited to, smartphones, tablets, wearable devices, Internet-of-Things (IoT), implantable medical devices (IMD), etc. Increasingly more people, especially the young generation, are getting used to the change of habits and lifestyle brought by such modern electronic devices. Smartphones and tablets are able to support most of the tasks people need in daily lives, such as web chat, online shopping, and entertainment. According to Pew Research Center [1], 77% of the U.S. population use a smartphone and 51% of Americans own a tablet. The wearable devices have also been welcomed in the consumer electronics market, especially those with fitness tracking function like smart watch and bracelet. IoT devices have a wide span of applications in both indoor and outdoor environments, ranging from the individual home, multi-floor building, up to a large city. IMDs are the particular type of wireless medical device that are installed into the patient’s body, to diagnose, monitor, or treat a variety of conditions, diseases and injuries. It is critical to ensure the security and robustness of these wireless devices, since the adversaries could launch either external wireless attacks from the open wireless channel or internal attacks through malware, to illegally gain access to the private data or the sensitive functionalities of the devices. In this dissertation, we investigate several unsolved security vulnerabilities in smartphones and devise effective solutions. We also design novel access control schemes for IoT devices and IMDs, respectively, to enhance the authentication and authorization processes.
1.1 Background

Although the manufacturers of modern wireless and mobile devices all keep security in mind when designing and building their products, the continuous reports of security breaches and bugs indicate that there is still a long way to go.

Smartphone security has been a hot topic over the past decade. Android and iOS are the two mobile operating systems (OS) that dominate the smartphone market. As for 2016 Q3, Android and iOS own a market share of 86.8% and 12.5%, respectively [2]. Android has been considered as less secure than iOS due to its open-source nature and that Apple has a more rigid vetting process over the official iOS app store. Despite the fact that dozens of security enhancements are made in each Android major version upgrade (e.g. Android 7 Nougat [3]) and that Android also releases monthly security updates to fix the security issues recently discovered [4], new software and hardware vulnerabilities at all security levels continue being reported. According to the annual report by the security firm F-Secure in 2017 [5], more than 99% of all malware designed for mobile devices targets Android devices. This does not mean that iPhones are perfectly protected. In March 2016, the first iOS Trojan AceDeceiver that is able to infect non-jailbroken iOS devices is reported by Palo Alto Networks [6]. AceDeceiver existed on the Apple App Store for eight months without being discovered and successfully bypassed Apple’s code review seven times. In July 2016, an iOS flaw is found that allowed malicious executable code to be sent via iMessage in Tagged Image File Format (TIFF) files [7].

Unlike smartphones that support a variety of functionalities, IoT devices and IMDs are embedded systems or micro-electromechanical systems (MEMS) for specialized purposes. There is no common security standards for IoT devices and IMDs. Instead, the individual manufacturers design and build their own devices including the mechanisms for security. This raises the risk of security breaches, as not all of the
manufacturers have a security expert in their research and development teams. For example, the vast internet outage happened in 2016 was caused by the Distributed Denial of Service (DDoS) attacks powered by a huge amount of unsecured IoT devices [8]. These IoT devices are infected and turned into a botnet by malware like Mirai [9] and IRCTelnet [10] using their factory default passwords. In addition, researchers found a major bug of the Philips Hue smart bulbs, which can be exploited by an attacker to remotely take control of the bulbs [11]. Such attack can spread by having the bulbs infect each other in a chain reaction, using just their built-in ZigBee wireless connectivity and physical proximity. The security risks of IMDs have also caused grave concerns. Halperin et al. [12] presented the vulnerabilities of a commercial implantable cardioverter defibrillator (ICD). Equipped with an oscilloscope and a software radio, they managed to reverse-engineer the ICD’s communications protocol and obtain the personal information of the patient and the ICD. Furthermore, they also launched active attacks to change the therapy settings and drain the battery more rapidly. Similarly, eavesdropping attacks and active attacks can compromise commercial glucose monitoring and insulin delivery system as well [13–15]. After reverse-engineering the communication protocol and packet format, they were able to impersonate the doctor and alter the intended therapy by replaying and injecting messages with a software radio. A security professional Barnaby Jack has also revealed serious security flaws in IMDs, and demonstrated how an adversary can remotely take full control of insulin pump, pacemaker and ICD [16]. Although the governmental agency U.S. Food and Drug Administration (FDA) is obliged to supervise and regulate the IMD industry, it only provides guidelines and recommendations for IMD security which are not legally binding [17,18]. There is no checking or verification of new IMD products (software and hardware) and their cybersecurity documentations by a trusted agency.
1.2 Motivation and Challenge

The security and privacy issues of smartphones, IoT devices, and IMDs are receiving considerable attentions from both the industry and academia. The researchers are actively seeking any vulnerability in existing wireless and mobile devices and designing new security schemes to enhance the protection on these devices. In this dissertation, we present our solutions on three mobile attacks that have not been well addressed, and two novel access control schemes for IoT devices and IMDs to prevent unauthorized access.

We choose Android system as the platform for our research on smartphone security, since Android is open-source so that we can customize it in order to include our defense schemes to a variety of malicious attacks. However, we must emphasize that the ideas of our countermeasures are effective regardless of the underlying mobile operating systems. Google’s Android Security Team has been devoted to solving all kinds of bugs in Android OS and hardware drivers. Instead of exploiting the intrinsic design flaws, we are interested in how malware can launch attacks using just the regular application programming interfaces (APIs). The major challenge is that benign apps may call these APIs as well, and we must identify the key differences in the usage patterns of the benign apps and malicious apps.

Specifically, we focus on the user interface (UI) attacks including the phishing attacks [19] [20] and the clickjacking attacks [21], as well as the attacks based on phone camera [22] [23]. Although the detection of web phishing attacks on PC browsers has already been very effective, the phishing detector module is removed from the mobile browsers in order to adapt to the resource constrained mobile phones. Hence, a phishing webpage that has been blocked by PC browsers (e.g., Google Chrome) can still be accessed on the mobile version browsers (e.g., Chrome for Android). Additionally, malicious mobile applications can also launch UI attacks such as the
phishing and clickjacking attacks, which are specifically for mobile platforms. In this dissertation, we investigate these mobile UI attacks, and design lightweight defense schemes that can be included into the mobile OS. Another type of mobile attacks we studied is the attacks based on phone camera. According to a report by Pew Research Center [24], 124,733 apps on Google Play Store (Android official app store) request the permission to take pictures and videos in 2014. We want to find out if a malicious app is able to stealthily use the camera without the phone user’s knowledge. Unlike the PC web camera, phone camera does not have a light-emitting diode (LED) indicator showing its working status. If possible, this would be a serious violation of user privacy.

Our second research focus is the security of IoT devices. Generally speaking, IoT devices are built to better connect us with the physical world, so as to greatly improve our lives. Now, we can find more smart appliances in indoor environments like home and office buildings. As a popular trend, the proximate IoT devices have been enabled to “talk” with each other, sharing the sensed data or even issuing commands (e.g, controller device or IoT hub) to each other to better serve the users. However, recently a group of researchers have demonstrated that they were able to hack the smart bulbs in a building, with a drone flying hundreds of meters away [11]. This attack warns us that an external adversary may launch attacks to the IoT devices within our homes. Hence, we propose to enhance the access control of IoT devices with an out-of-band secondary authentication, which can verify if the access requestor device is located inside the house or not [25].

The third piece of our work is on the access control of IMDs. Unlike the smartphones and IoT devices, IMDs are implanted in human body and are extremely limited in computational resources and battery. Meanwhile, IMDs are supposed to be accessible by any eligible physicians, which means the IMD and the IMD programmer (device specifically used to program the IMD) brought by the physician may not
have pre-shared secrets to encrypt their communications over the wireless channel. Therefore, we need to design a lightweight and secure access control scheme for IMDs [26].

1.3 Major Contributions

In this dissertation, we carry out the study on the security and privacy issues in modern wireless and mobile devices, including the already prevalent smartphones, the relatively new but quickly spreading IoT devices, and the highly constrained IMDs. Motivated by the vulnerabilities in these devices and the limitations of existing solutions, we present three types of attacks targeted at smartphones together with our countermeasures, and the novel access control schemes for IoT devices and IMDs, respectively. The contributions of this dissertation can be summarized as follows:

- We identified the weaknesses of the heuristics-based anti-phishing schemes that highly rely on the HTML (HyperText Markup Language) source code of web pages. Instead, we employ the optical character recognition (OCR) technique to extract the texts from the screenshot of a login interface, which are the true content displayed to the user. The novel lightweight anti-phishing scheme we propose, MobiFish, is capable of defending against phishing attacks on mobile web pages, apps, and persistent accounts. MobiFish can solve the essential problem of identity masquerade without reliance on HTML source code, search engine, or machine learning techniques. We implement MobiFish on Android smartphone and evaluate its effectiveness and overheads with extensive experiments.

- We comprehensively investigate the mobile clickjacking attacks, including the “after-attack” disguises to keep the clickjacking undiscovered after one successful attack which has not been considered in previous works. Additionally,
we explore a variety of clickjacking attacks, targeted on system apps, third-party apps, and other particular system UIs. The detection scheme we propose requires no user/developer involvement and is compatible with Android’s intrinsic system design as well as existing apps. We implement our defense scheme on Android smartphone. The experimental results show that it is effective and efficient.

- We also specifically study the camera-based attacks on smartphones, which have not been given enough attention by the research community. We discover two types of advanced spy camera attacks, the remotely-controlled real-time monitoring attack and the video-based passcode inference attacks. We tested them with several major mobile anti-virus apps, but none of them can detect any of the camera-based attacks we developed. We demonstrate that it is possible to recover simple password (e.g., 4-digit PIN) from the secretly captured video, which contains the user’s eye movement while entering the password. Finally, we propose an effective defense scheme to secure smartphone from all the camera-based attacks.

- To better enforce the access control among home IoT devices, we design a secondary authentication scheme over an out-of-band channel (light, acoustic, etc.), which can detect if the access requestor is a legitimate IoT device deployed in the same indoor environment or an external malicious device. We use Blockchain to record the relationship information of the home IoT devices and the result of the secondary authentication. We set up the experiments with commercial Blockchain and emulator devices Rasberry Pi. The performance of our proposed scheme is evaluated through experimental testings.

- We survey and analyze the current literatures for IMD access control, based on the access control architecture, access control model and whether they are
able to cope with the special use cases. To resolve the limitations of existing schemes, we propose a novel proxy-based IMD access control scheme which can greatly alleviate the computational overhead and power consumption of IMDs. The proxy communicates with the IMD programmer through the audio cable, so that it can avoid the passive and active attacks over the wireless channel. The ciphertext-policy attribute-based encryption (CP-ABE) is employed to provide a fine-grained access control over the qualifications of the programmer operator. We implement our scheme on real emulator devices. The evaluation results show that the proposed scheme is lightweight and effective.

1.4 Dissertation Overview

In this dissertation, we first briefly introduce the background, the motivation and challenges, and our major contributions. In Chapter 2, we present our work on mobile phishing attacks and the detection schemes. In Chapter 3, we describe our study on mobile application clickjacking attacks and the details of our countermeasure. In Chapter 4, we investigate the camera-based attacks on smartphones and our proposed defense scheme. In Chapter 5, we design an out-of-band secondary authentication scheme for home IoT devices. In Chapter 6, we propose a novel proxy-based and fine-grained access control scheme for IMDs. The conclusion and our future directions are given in Chapter 7.
MOBILE PHISHING ATTACKS

Smartphone security has been attracting particular attention and interest in recent years. Mobile phishing is one of the most dangerous attacks, due to the hardware limitations of mobile devices and the mobile user habits. In this chapter, we provide a comprehensive study on the security vulnerabilities caused by mobile phishing attacks, including the web page phishing attacks, the application phishing attacks, and the account registry phishing attacks. Existing schemes designed for web phishing attacks on PCs cannot effectively address the various phishing attacks on mobile devices. In this chapter, we propose MobiFish, a novel automated lightweight anti-phishing scheme for mobile platforms. MobiFish verifies the validity of web pages, applications, and persistent accounts by comparing the actual identity to the claimed identity. MobiFish has been implemented on a Nexus 4 smartphone running the Android 4.2 operating system. We experimentally evaluate the performance of MobiFish with 100 phishing URLs and corresponding legitimate URLs being impersonated, as well as phishing apps. The results show that MobiFish is very effective in detecting phishing attacks on mobile phones.

2.1 Introduction

Phishing attacks aim to steal private information such as usernames, passwords, and credit card details by way of impersonating a legitimate entity. Although security researchers have proposed many anti-phishing schemes, the threat of phishing attacks is not well mitigated. On the one hand, lots of phishing sites expire and revive
rapidly. According to the Anti-Phishing Working Group (APWG), the average time that a phishing site stays online is 4.5 days [27]. Cranor et al. [28] even found that, sometimes, it is on the order of hours. On the other hand, phishing attackers keep improving their techniques so that their new attacks are able to circumvent existing anti-phishing tools.

Mobile phishing is an emerging threat targeting mobile users of financial institutions, online shoppers, and social networking companies. In 2012, researchers from Trend Micro found 4,000 phishing URLs designed for mobile web pages [29]. Although this number takes up less than 1% of all collected phishing URLs, it highlights that mobile platforms have become new targets of phishing attacks. Notice that mobile users could also be spoofed by conventional phishing web pages (designed for PC browsers) when browsing with their phones. The trend of launching phishing attacks on mobile phones may be attributed to the hardware limitations such as the small screen size, the inconvenience of user input and application switching, the lack of identity indicators, mobile user habits and preferences, etc.

Almost all phishing attacks on PCs are in the form of bogus websites. Nowadays, with browsers powerful enough to support all kinds of Internet services, people are accustomed to online banking, online shopping, online socializing, etc. They are familiar with being requested to provide, and subsequently providing private information and credentials to websites. Current phishing detection schemes can be roughly divided into two categories: heuristics-based schemes and blacklist-based schemes. Blacklist-based schemes can detect only those phishing sites that are in the blacklist, and cannot detect zero-day phishing attacks such as those that have only appeared for days or hours. It is possible that new phishing sites may have already stolen user credentials or have expired before being added into the blacklist. Heuristics-based schemes largely depend on features extracted from URL and HTML source code, and then other techniques such as machine learning are used to determine
the validity. However, we find that the features extracted from HTML source code could be inaccurate, and phishing sites can easily bypass those heuristics.

Moreover, browsers have many practical features and convenient functions abandoned or truncated during their adaptation to hardware-constrained mobile platforms; this results in an unpleasant experience for users. To improve their services, most well-known enterprises have published mobile applications (apps) for major mobile platforms. This sheds new light on phishing scams: some phishing attackers develop fake apps or repackage legitimate apps, and then upload these phishing applications to unofficial app markets. Once the attack succeeds, the victim’s credentials will be sent to the phishing server. Phishing apps are even harder to detect than phishing web pages, since for web pages, we are able to judge the destination of form-data from HTML source code (action attribute in the form tag). But for mobile apps, there is no way to check if user credentials are sent to the legitimate authentication server or the attacker’s server. Hence, phishing attacks on mobile phones are more complicated than those on PCs. It is important to design effective mobile phishing defense schemes for both web pages and applications.

Besides, we further discover a specialized form of phishing attacks which target at the persistent account registry function of mobile OSs. Since the malicious apps that have created a persistent account interact with the user using a separate login interface (not the app’s login interface), we also need to design a defense scheme specific to account registry phishing attacks.

2.2 Background

In this section, we conduct a comprehensive analysis on the factors that make mobile users vulnerable to phishing attacks, from both the objective perspective of hardware limitations and the subjective perspective of mobile users themselves.
2.2.1 Hardware Limitations

Due to the small size of phone screens and limited computational power, browsers in mobile systems have to remove or degrade some features to make more space for web contents and maintain a smooth user experience (e.g., loading speed). Unfortunately, the security-related functionalities are among those missing features. As a result, phishing web pages that could have been detected and blocked on PC browsers may still be accessible from mobile browsers.

In addition, the UI of mobile browsers is also simplified, which could instead help phishing sites to bypass user inspection. To accommodate the small phone screen size, most mobile browsers have to remove the status bar and hide the URL bar once the web pages finish loading. Even during the loading process, long URLs are truncated to fit the browser frame. Since the ability to read and verify URLs is crucial in detecting phishing attacks, partial URL (especially a URL with only partial domain name displayed) would certainly increase the user’s risk of being spoofed by the phishing attacks. For example, Figure 2.1(a) shows the URL bar with only a partial domain name when loading the “Bank of America” website. This could lead to a successful phishing attack if users are convinced by the partial URL and submit their credentials, while the full URL turns out to be “https://secure.bankofameri.com” or “https://secure.bankofamerica.com.phishing.com”. Such tricks would fail if the entire URL (or at least the full domain name) is displayed. One possible way by which a
user can view the complete URL is to click the address bar and manually scroll all the way to the end. Another way is to view the actual destination of a link, which can be invoked by holding the link for about two seconds. Though the destination URL is also partially presented as in Figure 2.1(b), it can display the domain name with as many as 31 characters, instead of 19 characters in the URL bar. Since the full domain names of the login pages are no longer than 30 characters for most legitimate sites, checking the destination allows users to detect phishing sites more quickly.

Moreover, for some legitimate sites, their domain names could be easily mimicked by replacing the letters. For example, it is hard to distinguish ‘l’ from the capital ‘i’ because mobile browsers display them both in vertical slash shape (e.g., Figures 2.2(a) and 2.2(b)). In Figure 2.2(c), we list both ‘l’ and capital ‘i’ together and find that their small difference in height is difficult to discern by human eyes. For this kind of letter replacement phishing attacks, even attentive and observant users who always check the entire URL (domain name) are likely to be fooled.

Besides, the lack of identity indicator is another issue in mobile phishing. Unlike the URL bar in browsers, there is no straightforward indication of identity available for mobile apps and persistent accounts. Users may manually look up the task list.
to check the identity of the running apps. However, it is not possible to verify which app a persistent account has been bound to.

2.2.2 User Habits and Preferences

The habits and preferences of mobile users further increase the vulnerability to mobile phishing attacks. During the past few years, touch screen smartphones have become dominant in the mobile phone market. However, typing on a virtual keyboard is not as easy as on a physical keyboard due to the lower input accuracy, particularly when walking or sitting in a moving vehicle. Because of that, it is tempting to follow links in web pages or e-mails rather than typing the links manually. Another factor is that, on smartphones, switching among applications or even shifting to other pages within a browser is more complicated and tedious than being performed on a PC. Users who value convenience usually prefer to follow links from other applications [30]. In addition, phishing attacks can succeed because users have become accustomed to entering their credentials in familiar, repeated login interfaces. If users frequently encounter legitimate links whose targets prompt them for private data, then they get used to reflexively supplying the requested data [31].

2.3 Mobile Phishing Attack Models

This section presents three types of phishing attack models: mobile web page phishing attack model, mobile app phishing attack model, and mobile account registry phishing attack model.
2.3.1 Mobile Web Page Phishing Attacks

Web page phishing detection has been widely studied and applied in PC browsers. Blacklist-based matching and Heuristics-based detection are the two major existing methods used for web page phishing detection. The blacklist method is to search a suspicious site in a list of reported phishing sites. Although it can achieve high accuracy at the cost of human verification, the delay in updating the blacklist would greatly degrade its effectiveness. Specifically, blacklist-based methods cannot defend new phishing sites that have not been listed, such as zero-day phishing attacks.

Heuristic detection methods are based on features extracted from URL and HTML source code, and often work with the assistance of search engine or machine learning techniques. These features are summarized from previously reported phishing sites. However, a new phishing site may not have these features at all because each feature only appears in some of the phishing samples. This means carefully constructed phishing sites that remove all suspicious features are able to bypass the heuristics-based detections (this is why heuristics-based approaches cannot achieve a 100% detection rate).

Besides, we find that information extracted from HTML source code may not be able to reflect the web page displayed to the user. This is because attackers can add text, images, and links into HTML source code; meanwhile, they can also make “undesirable” content disappear from a web page by simply changing its size or covering it with other images. Therefore, features like word frequency, brand name, and company logo could be easily manipulated. For example, Figure 2.3(a) shows the real Ebay mobile login page. We copy the code of the original site and migrate it to our website. We also upload the image components to our website and change the links to the corresponding places within our website, especially its form-data submission URL. Then the code segment in Figure 2.3(c) is added into the source
code. However, the tampered web page (Figure 2.3(b)) looks exactly the same as the official Ebay site. No user will suspect its validity without looking at its URL. Meanwhile, phishers can insert as many “bugs3” as needed into the HTML source code to obfuscate conventional identity extractors. The large number of “bugs3”s extracted are able to convince the identity extractor that this web page claims to be “bugs3” instead of “Ebay”. As a result, anti-phishing heuristics would fail since the phishing web page indeed belongs to the “bugs3.com” domain. The title of a web page is not visible unless the user clicks the page menu icon and switches to an overview of the opened page list, which means the title could also be replaced by “bugs3” to enhance the consistency of HTML source code. Hence, HTML source code is not a reliable source for phishing web page detection. We seek for a new approach
to extracting the identity of a web page.

2.3.2 Mobile Application Phishing Attacks

Phishing attacks based on applications are quite uncommon in PCs, but are disturbing problems on mobile platforms. According to Felt et al. [32], the four high-risk phishing attack models with a “common” prevalence level and a “perfect” accuracy level are all associated with mobile applications that impersonate legitimate apps. Application-oriented phishing attacks can be further categorized into two types based on the way in which they are launched: Some phishing applications attempt to hijack existing legitimate apps. These phishing apps continuously perform task polling, and launch themselves as long as they detect the launch of the target apps. As a result, the fake login interface layers over the top of the real one, and the phishing app “appears” to be the target app. Mobile users do not know what has happened since everything is accomplished during a single window switching process. One possible way to solve this is to check the identity of the current foreground app from the task list, though normally no user does that. Another type of phishing app directly shows up as the target app. This may occur when a user downloads fake apps from unofficial app markets.

Despite the various methods of stealing user credentials, the essential attack pattern must end with the transmission of credentials to the attacker. Hence, runtime monitoring and blocking the communication of the suspicious apps can effectively defeat the app phishing attacks.

2.3.3 Mobile Account Registry Phishing Attacks

The Android system provides a centralized account registry and management function which allows the phone users to log in to their online accounts (e.g., Google,
Facebook, Twitter, etc.) in a once-for-all manner. We call this function “Persistent Account Registry”. It is very convenient in that phone users will not be bothered to enter credentials each time they launch a social networking app or a financial institution app. Since most phones are in personal use and serve only the phone owner, the user’s privacy can be guaranteed by simply setting an “overall” screen locking password. In the Android system, the persistent account registry can be accessed through “Add account” in the “Settings” directory, as depicted in Figure 2.4(a).

Figure 2.4(b) shows a list of applications that support persistent account registry. Basically, any app can register itself to show up in the account list as a new account type. The app should have an authentication service which is bound to the AccountManagerService of the Android system. This service must specify the following intent filter (i.e., with action ACTION_AUTHENTICATOR_INTENT) and corresponding metadata tags in the manifest file.

```xml
<intent-filter>
  <action
    android:name="android.accounts.AccountAuthenticator" />
</intent-filter>
<meta-data
  android:name="android.accounts.AccountAuthenticator"
  android:resource="@xml/authenticator" />
```
To make an app appear properly in the account list, the *android:resource* attribute in the metadata will point to an *xml* resource file that contains at least the following 3 attributes: *android:accountType* is the name of the new account type which uniquely identifies this account/app; *android:icon* and *android:label* are the icon and label displayed in the account list. Besides, *android:smallIcon* is used in the contact application’s tab panel, and *android:accountPreferences* points to an *xml* hierarchy which contains the *PreferenceScreens* that can be invoked.

```xml
<account-authenticator
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:accountType="typeOfAuthenticator"
 android:icon="@drawable/icon"
 android:smallIcon="@drawable/miniIcon"
 android:label="@string/label"
 android:accountPreferences="@xml/account_preferences" />
```

When a certain account type is selected to be added by the phone user, the authentication service of the corresponding app will be invoked and it will create an *authenticator* instance. The *authenticator* class extends the *AbstractAccountAuthenticator* class, overrides the *addAccount* method, and will launch the login interface in which the user is prompted to enter the credential.

We found that persistent account registry is vulnerable to phishing attacks as well. The icon and label displayed in the account list are defined by corresponding applications, and are not necessarily the same as those used in the main menu. That is to say that any third party app could register itself as another entity in the account list. For example, a malicious game app may pretend to be an app of a social networking company or financial institution.

Our demo app shows up as a “Fake Twitter” app in the account list (as in Figure 2.4(b)), and its login interface (Figure 2.5(b)) looks exactly the same as the legitimate “Twitter” app (Figure 2.5(a)). The difference is that when a user clicks the “Sign In” button, the credentials will be sent to the attacker instead of the
Twitter authentication server. In the app phishing attacks, we have dealt with an indistinguishable fake login interface using AppFish scheme. However, the account phishing attack is stealthier. When the login interface is present to the user, the user does not know which app it truly belongs to. As in Figure 2.5(c), even if a phone user checks the task list, Settings is the only app in there. The untraceable nature of the account registry interface leaves it out of the protection of AppFish.

In the design of an account phishing app, a practical issue has to be solved: if there appears to be two “Twitter” apps in the account list, the user could immediately know the abnormality. To avoid duplicates, the malicious app should perform a scan of all installed apps before registering a phishing account. In fact, the phishing app could incorporate multiple sets of authenticators (including the authentication service, login interface, etc.), each for one specific account type. This means that app has the potential to masquerade as multiple legitimate apps. Each time, it only chooses to be one of the apps that have not been installed on the victim device. The polling of the installed apps can be easily accomplished using PackageManager, and it is stealthy in that no permission is required. However, if all the target apps have been installed, no account phishing attack will be launched.
It may sound impossible to load the xml resource file (where the account icon and label are defined) dynamically, since it is specified explicitly in the manifest file and will only be read once (during the installation process). However, we have discovered a tricky bypass to this limitation. As the xml resource file is referenced in the metadata tag of the corresponding authentication service, the account information contained in the xml file will not be registered if that service is “disabled”. Initially, we set the android:enabled attribute of all authentication services to false. As soon as the account phishing app is installed, it scans the installed apps and decides which account to impersonate. Then, it calls the setComponentEnabledSetting method of PackageManager which can override the “disabled” state that has been set by the service in manifest file. For example, if the legitimate Twitter app is missing in the phone, the phishing “Twitter” authentication service will be enabled and the fake “Twitter” account will be registered in the account list during runtime. Furthermore, the account phishing app needs to periodically check the installed app list. If duplicates are detected (user later installed the legitimate app), it could switch to another account type/app that has not been registered-installed.

Note that although Android uses several permissions to regulate the persistent accounts (e.g. ACCOUNT_MANAGER, AUTHENTICATE_ACCOUNTS, MANAGE_ACCOUNTS, USE_CREDENTIALS, and GET_ACCOUNTS), they are either specified for the authentication process, the retrieving of the existing accounts information, or are reserved only for system apps. Our demo app “Fake Twitter” (as shown in Figure 2.4(b) and Figure 2.5(b)) can appear in the account list, present the fake login interface, and request user credentials without any account-related permission.
2.4 Overview of MobiFish Scheme

2.4.1 Motivation

Phishing attackers take fancy tactics to direct victims to their phishing sites or applications, which masquerade as trustworthy entities. The key of solving the phishing problem is to find the discrepancy between the identity it claims to be and the identity it actually is. We have shown that HTML source code is not a reliable clue to find the claimed identity of a phishing site. As an alternative, we should focus on the screen presented to the user since users are directly spoofed by what they see. Besides, existing anti-phishing schemes cannot detect app phishing attacks and account registry phishing attacks. Thus, there is a strong need for an effective defense scheme against the phishing attacks on mobile platforms.

2.4.2 Identity Extraction

As discussed above, the screen presented to mobile users should be the exact place where the claimed identity is extracted from. It turns out that a good way to capture the screen content is to take a screenshot. There are two common observations that lead us to believe that screenshots can work well in identity extraction and verification. The first is that most login interfaces of legitimate mobile sites and apps are very simple. The entire login page, or the majority of the page, can be captured in one screenshot. Another observation is that the brand names and the company logo (identity) locate at apparent places in the login page, which can be easily captured and extracted from the screenshot. Screenshots can be used for the phishing detection in both web pages and applications. Since the source code of the apps is not available, there is no way to acquire the content of an app login interface other than taking a screenshot. Then, to obtain the claimed identity, the OCR technique is utilized to
The actual identity of a mobile web page can be obtained from the second-level domain name (SLD). Most well-known enterprises use their brand names as the SLD of their official websites. This can be best illustrated by Bank of America (BoA). BoA uses the entire brand name as the SLD despite its length. In special cases that brand names are not exactly the same as the SLD, e.g., “AT&T” which contains a symbol in the brand name, all content-based schemes will fail due to the mismatch of brand name “AT&T” and SLD “att” (for legitimate URLs, special symbols are usually not included in domain name). However, such inconsistencies can be easily solved with a mapping whitelist, in which the brand name “AT&T” is mapped to the SLD “att” before the identity verification, or vice versa. Again, due to the unknown source code, the actual identity of mobile apps cannot be decided until the transmission of user credentials happens. The checking of the suspicious apps must be cleared before they are allowed to transmit.

Besides, to detect malicious apps launching account phishing attacks, we first need to ensure that the identity shown in the main menu (app name) and the identity displayed in the account list (account label) are consistent. But even if this condition is satisfied, it could still be a repackaged legitimate app. We need to find out its actual identity by tracking where the credentials are sent to, as the same in AppFish scheme.

2.4.3 OCR Techniques

Optical character recognition (OCR) is the mechanical or electronic conversion of an image to machine-encoded text. According to previous works, it is valuable for phishing detection as long as high-quality OCR solution is used [33]. OCR has been utilized to extract text from simple text-only logo in [34]. GoldPhish [35] uses
optical character recognition to read text from a web page (specifically the company logo). We believe that the OCR technique could achieve better performance on mobile phones because phones have a smaller screen size, and a relatively higher pixel density.

We deploy the OCR technique into the mobile platform and show that it achieves better performance and effectiveness on mobile phones by real experiments. The tool we use is Tesseract [36], which is one of the most accurate open source OCR engines, and supports over 60 languages. Our testing uses a Thinkpad T420 laptop (2.40GHz, 4GB RAM) with a pixel density of 131 dpi and a Google Nexus 4 smartphone (1.5GHz, 2GB RAM) with 320 dpi.

We open the Ebay mobile login page in both mobile and PC browsers, each capturing a screenshot (as shown in Figures 2.3(a) and 2.6(a)). Tesseract is used to extract text from the screenshot taken on the mobile phone while Microsoft Office Document Imaging (MODI) is used for the screenshot of PC browser (this tool is used...
in GoldPhish [35]). The results are given in Figures 2.6(b) and 2.6(c), respectively. We find that Tesseract extracts all words correctly from the mobile screenshot, except the “sign in” in the dark blue button. The performance of MODI on PC is not as good as Tesseract, because MODI not only missed the dark blue button, but also missed the word “ebay” in the top-left logo. This example also shows the ability of Tesseract to deal with various styles and fonts of text in company logos. Moreover, the OCR extraction on mobile phones only takes 1.6 seconds, while on PC, the time is 4.5 seconds. To mitigate the influence of different OCR engines, we also extract the screenshot of a PC web page using Tesseract (Windows version). Although it takes only 1.5 seconds, the accuracy is much worse because as many as 10 words are extracted wrong, including the Ebay logo (Figure 2.6(d)).

The above tests show that OCR achieves higher accuracy and efficiency on mobile platform. It plays an important role in the identity extraction module of our mobile anti-phishing scheme.

2.5 The MobiFish Scheme

In this section, we present an automated lightweight scheme for mobile phishing defense named MobiFish. MobiFish consists of three major components named WebFish, AppFish, and AccountFish, which are designed to protect mobile web pages, applications, and persistent accounts, respectively.

2.5.1 The WebFish Scheme

The work flow of WebFish is given in Figure 2.7. As we can see, the defense scheme is initiated with URL loading. When a browser attempts to load a web page, WebFish first scans its URL to see whether the domain name is an IP address. Legitimate websites always use domain names as a verification of their identities,
while phishers are likely to use an IP address to disguise their fake identities. Next, WebFish obtains the HTML source code of the loading page, and checks if there is any form in that page. Like legitimate login pages, phishing web pages also need a form with an input tag which allows user to enter (confidential) information and then submit. WebFish checks the existence of forms so that not every page has to go through the checking. However, the core module of identity extraction does not rely on any part of HTML source code. If a form is found, WebFish starts the identity extraction and verification. On one hand, it extracts the SLD from the URL, which represents the actual identity of the site. Then the SLD is indexed in the Mapping White-List (MWL). If it matches any of the SLD-Brand name records in the MWL, the original SLD is replaced by the corresponding brand name. On the other hand, it calls the screencap native function to take a screenshot of the login page and extract the text with the OCR tool. Note that the URL shown in the URL bar may also be captured into the screenshot, and it should be removed from text before identity verification, as it contains the actual identity (SLD) of the site. The existence of a
URL bar in the screenshot can be determined by whether the first line contains one of the top-level domain names (e.g. gov, edu, com, and org). To further speed up the detection process, we search for sensitive terms such as “username”, “password”, and “credit card number” in the text. If not found, the form may be just used for search or general data input purposes, and the page is marked as safe directly. Otherwise, the last step is to search the SLD in the text. If not found, it is marked as a phishing site. WebFish shows a notification window to the user, indicating the high possibility of a phishing attack, along with the URL of the suspected web page.

Our design is based on the assumption that, if the domain name of the phishing site appears in the fake login page of a legitimate entity, the user can immediately discern the difference and check the URL to verify the validity of this web page. This is reasonable since as far as we know, no phishing site uses common terms in login pages like “sign”, “username”, “password”, or “welcome” as the SLD. Legitimate mobile login pages are made very simple and clear. It is highly unlikely for these well-constructed and well-maintained web pages to have strange words (SLD of phishing sites) appear in them. Thus, users would become alerted if a web page contains text different from the brand name or common login terms. If the attacker adds the phishing domain name in tiny font size to prevent the user from noticing, then the OCR is not able to recognize it either, and WebFish will still mark it as a phishing site. The key feature for WebFish to detect a phishing URL is that the SLD is not among the text extracted from the screenshot of the login page.

### 2.5.2 The AppFish Scheme

The work flow of AppFish is shown in Figure 2.8. AppFish maintains a database called Suspicious App Set (SAS), which contains the profiles of untrusted apps including the user ID (Uid), the launching time, and the screenshot text. Users
can add the apps they suspect into SAS, and only apps listed in SAS are under the monitoring of AppFish. These apps can be characterized as:

1. Specified for one company. This is to ensure that the app only contacts the company’s official or affiliated (partners) servers. The domain names of the collaborators are collected and added into the SAS profile in advance. Owning multiple domains often happens to websites that need extra storage. For example, we find in our testing that Facebook may request data from domains like fbcdn.net and akamaihd.net. This is because Facebook uses them as a content delivery network (CDN). The substantial amount of photos generated by Facebook users are uploaded to akamaihd.net instead of facebook.com. Whenever a user wants to view a photo, the request is actually sent to the nearest akamaihd server.

2. Require user sign in. There are lots of apps that do not need users to login, like browsers and apps for news, music, maps, etc. In these cases, phishing attacks would not happen at all. For browsers, web page login is protected by WebFish.

The AppFish defense scheme works in two phases: launching phase and authentication phase. In the launching phase, AppFish obtains the name of each launching application and searches for it in SAS. If found, the logging process begins,
in which AppFish takes a screenshot of the login interface and extracts the text with the OCR tool. Then, the text along with the application Uid and the launching time are logged into the profile of that app. After the user has entered the credentials and clicks the “sign in” button, the authentication phase begins. Legitimate applications (like Facebook and Twitter) usually send the user’s credentials to a remote server for authentication via HttpGet/HttpPost. Once the credentials are verified, the application loads data belonging to that account. On the contrary, phishing apps are not able to load user data, and the only trick they can play is to tell the user that he or she has entered the wrong password. However, after two or three times, most users will suspect the validity of the application, and will uninstall it. Hence, a phishing app is able to send out the user credentials only during the period from submission to uninstallation. AppFish monitors the possible paths for a phishing app to transmit data to the outside world, which include HttpGet/HttpPost, socket, Short Message Service (SMS) and email (email is also based on socket). If an application uses any of these means to send out information, AppFish checks whether it is one of the suspects in the SAS. If confirmed, http connections (HttpGet/HttpPost) are filtered while other communications (socket/SMS) are blocked. For all URLs the suspicious app requests to connect with, AppFish ensures that the SLD name appears in the screenshot text or the affiliated domain names stored in the SAS profile. Meanwhile, socket and SMS functions are blocked for a certain amount of time $T$, which should be long enough for the user to notice the abnormality and uninstall the malicious app. Thus, for phishing applications, they will not be able to send out credentials before being removed by the user.

Note that we have to search the SLD within the text extracted from the login page. The reason why extracted text (instead of the application name) is used is that: for phishing attacks based on task interception (hijacking), the phishing app can have the same app name as the SLD of the phishing server, while it can pop up a fake login
page in another entity. For instance, a mobile user downloads a phishing app named “abc”, due to its tempting fancy functions. However, this phishing app could pop up a fake Facebook login interface as soon as the legitimate Facebook launches. Once the user is spoofed, the app “abc” immediately sends the credentials to the phishing server “abc.com”. In this example, the foreground (fake) application name “abc” is the same as the SLD of phishing server, but it cannot be found in the “Facebook” phishing login interface.

2.5.3 The AccountFish Scheme

The unlimited registry of persistent accounts among third-party apps poses a huge threat to mobile user privacy and account security. According to Yahoo Aviate’s collected data in 2014 [37], the average number of apps installed for an Android user is 95. Users may not remember clearly what app has been installed, and hence are vulnerable to account registry phishing attacks. We propose AccountFish to defend against the phishing attack targeted at persistent accounts. The work flow of AccountFish is described in Figure 2.9.

The account registry phishing attacks can be classified into three types, based on the identities the malicious app appears to be in the main menu and the account list. In the type A attack, the malicious app appears to be a different app to the target account (e.g., a game app registers a Twitter account). In the type B attack, the malicious app does not appear in the main menu at all. In the type C attack, the malicious app directly shows up as the target app (e.g., repackaged app), which means that they will have the same application name shown in the main menu as the account name that appears in the account list.

The idea of the detection mechanism for the type A and type B account phishing attacks is to compare the app name in the main menu and the account label in
the account list. As mentioned before, the malicious app can dynamically register and change the account information. AccountFish should be able to inspect the registration of accounts in runtime, which can only be accomplished by modifying the Android source code. Specifically, we modified the parseServiceAttributes method of AccountAuthenticatorCache class so that the account label is extracted and compared with the host app name each time an account is being added (before the corresponding authentication service is called). If the app name and the account label are inconsistent, that app is highly likely to be a malicious app. But there are a few legitimate apps whose account labels are not exactly the same as corresponding app names (e.g., “Firefox Sync” in Figure 2.4(b)); we solve this issue using an account mapping white list (AMWL) that contains all the “inconsistent” legitimate apps. For suspicious apps that are not in the white list, a warning containing the app and account names will be issued instantly. The apps in the AMWL and those having consistent names will go through further checks (type C attack detection).

Figure 2.9: The Work Flow of AccountFish
The mechanism we use to detect the type $C$ attack is similar to AppFish, because the decision cannot be made until the transmission of credentials happens. The difference is that a screenshot is unnecessary in AccountFish: the “add account” event is not available to other third-party apps, so the account login interface cannot be hijacked. We add a hook in the `bindToAuthenticator` method of `AccountManagerService`, which can catch the user click event for adding a specific account and launch the communication monitoring mechanism. Specifically, the app name (account label) will be directly used to filter the outgoing Http connections (HttpGet/HttpPost): only URLs with the same SLD as the app name are permitted. Meanwhile, other communication channels of the suspected app (socket/SMS) are blocked for a certain amount of time $T$, which is long enough for a user to notice the abnormality and uninstall the malicious app.

2.6 Implementation and Performance Evaluation

We implement MobiFish on a Google Nexus 4 smartphone running the Android 4.2 operating system. We modify the source code of the Android system so that it is able to support MobiFish. The MobiFish scheme could be applied to other mobile platforms as well. To evaluate the effectiveness and performance of MobiFish, we conduct experiments for WebFish, AppFish and AccountFish, respectively.

2.6.1 Experiments with WebFish

In the process of evaluating WebFish, we were not able to collect enough phishing web pages specified for mobile platform. Instead, we randomly picked up 100 phishing URLs from PhishTank.com in 2013. Although all the phishing URLs have been blocked by PC browsers like Chrome, they are accessible through mobile browsers (including both Android’s built-in browser and Chrome for Android). This fact
Figure 2.10: Experiments with WebFish on Mobile Phishing Login Page

highlights the significance of WebFish, which can provide web phishing defense for mobile OSs. In our experiments, WebFish can effectively mark all the phishing URLs as dangerous, and can warn the user. Figures 2.10(a) and 2.10(c) show that WebFish displays alertness because it is not able to find the SLD inside the mobile phishing login pages of “Bank of America” and “AT&T”.

Meanwhile, we have two observations for the conventional (PC) phishing web pages. First, a large number of them are in high similarity to their legitimate counterparts, and the brand names or company logos are close to the input forms.
Second, when loading large conventional web pages, mobile browsers often display the area that contains the input form instead of displaying an overview of the entire web page. Figures 2.11(a) and 2.11(c) show the phishing login pages of Yahoo and PayPal. As we can see, the brand name Yahoo and PayPal logo appear more than once in the input form area that is presented to the user. Both of them are reported as phishing sites since WebFish cannot find their SLD in the screenshot.

To evaluate the performance of WebFish on legitimate web pages, we use the URLs
<table>
<thead>
<tr>
<th>Website</th>
<th>Phishing Samples</th>
<th>Phishing Feature Found</th>
<th>Legitimate Mobile Login SLD</th>
<th>Verification of Legitimate URLs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon</td>
<td>6</td>
<td>100%</td>
<td>amazon</td>
<td>✓</td>
</tr>
<tr>
<td>AOL</td>
<td>3</td>
<td>100%</td>
<td>aol</td>
<td>✓</td>
</tr>
<tr>
<td>Apple</td>
<td>5</td>
<td>100%</td>
<td>apple</td>
<td>✓</td>
</tr>
<tr>
<td>AT&amp;T</td>
<td>2</td>
<td>100%</td>
<td>att (MWL)</td>
<td>✓</td>
</tr>
<tr>
<td>Bank of America</td>
<td>10</td>
<td>100%</td>
<td>bankofamerica</td>
<td>✓</td>
</tr>
<tr>
<td>Barclays</td>
<td>4</td>
<td>100%</td>
<td>barclays</td>
<td>✓</td>
</tr>
<tr>
<td>Chase</td>
<td>5</td>
<td>100%</td>
<td>chase</td>
<td>✓</td>
</tr>
<tr>
<td>Citi</td>
<td>4</td>
<td>100%</td>
<td>citibank (MWL)</td>
<td>✓</td>
</tr>
<tr>
<td>Ebay</td>
<td>8</td>
<td>100%</td>
<td>ebay</td>
<td>✓</td>
</tr>
<tr>
<td>Facebook</td>
<td>5</td>
<td>100%</td>
<td>facebook</td>
<td>✓</td>
</tr>
<tr>
<td>Hotmail</td>
<td>2</td>
<td>100%</td>
<td>live (MWL)</td>
<td>✓</td>
</tr>
<tr>
<td>HSBC</td>
<td>8</td>
<td>100%</td>
<td>hsbc</td>
<td>✓</td>
</tr>
<tr>
<td>Microsoft</td>
<td>1</td>
<td>100%</td>
<td>live (MWL)</td>
<td>✓</td>
</tr>
<tr>
<td>NAB</td>
<td>1</td>
<td>100%</td>
<td>nab</td>
<td>✓</td>
</tr>
<tr>
<td>NatWest</td>
<td>3</td>
<td>100%</td>
<td>mwlolb (MWL)</td>
<td>✓</td>
</tr>
<tr>
<td>PayPal</td>
<td>12</td>
<td>100%</td>
<td>paypal</td>
<td>✓</td>
</tr>
<tr>
<td>Vodafone</td>
<td>4</td>
<td>100%</td>
<td>vodafone</td>
<td>✓</td>
</tr>
<tr>
<td>Wells Fargo</td>
<td>7</td>
<td>100%</td>
<td>wells Fargo</td>
<td>✓</td>
</tr>
<tr>
<td>Yahoo</td>
<td>10</td>
<td>100%</td>
<td>yahoo</td>
<td>✓</td>
</tr>
<tr>
<td>Total:</td>
<td>100</td>
<td>100%</td>
<td>Tot: 19</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 2.1: Summary of Tested URLs

of the corresponding official login web pages for comparison. Figures 2.10(b), 2.10(d), 2.11(b) and 2.11(d) are the official mobile login pages; WebFish successfully verifies the validity of these pages and no warning is generated. WebFish’s ability to verify the legitimate AT&T web page shows that the Mapping White-List (MWL) scheme works for company websites with different brand names and SLDs. The 19 corresponding legitimate mobile login web pages can prove WebFish’s ability in verifying legitimate web pages. Table 2.1 summarizes the testing results of phishing URLs (Columns 1, 2, 3) and legitimate URLs (Columns 4, 5). The “MWL” behind a legitimate SLD name means that MWL is used to convert the SLD to the brand name.

Table 2.1 shows that: (1) WebFish is able to find key features of phishing web pages for all tested phishing URLs; and (2) WebFish achieves a 100% verification rate of legitimate URLs. The results demonstrate the effectiveness of WebFish in
detecting mobile phishing sites.

2.6.2 Experiments with AppFish

By the time we conduct experiments with AppFish, there are only a few reported phishing applications, and none of them is available online. To test the effectiveness of AppFish, we develop two sample phishing applications. Figure 2.12(a) shows the login interface of the fake Facebook apps we developed. Most users are not able to discern its difference from the legitimate Facebook app. Our first phishing application appears as a “repackaged” Facebook app. The second one hijacks the real Facebook app. It can cover the real Facebook login interface in a single window switching slot, hence the user cannot notice that in fact two apps have been launched. When the user clicks the “Log In” button, the fake apps send the credentials to us by HttpGet, HttpPost, socket, SMS, and email, respectively. Meanwhile, a notice window is displayed, informing the user of an incorrect password, and prompts another try. But when AppFish is running, it is able to block all the transmissions and warn the user about the phishing attempts. Figure 2.12(b) (lower part) shows the warning
2.6.3 Experiments with AccountFish

The persistent account registry attack is a new class of phishing attacks that we have discovered. As far as we know, there is no such phishing app reported. Hence, we evaluate the effectiveness of AccountFish against the three types of account phishing attacks using the demo apps we developed.

Figure 2.13(a) presents a type A account phishing attack, in which a “Greedy Snake” app intends to register a Twitter account. Figure 2.13(b) shows a sample type B attack, where the suspicious “Game Center” app wants to register a Twitter account while hiding itself from the main menu. Both attacks are successfully detected when trying to register the accounts whose labels are inconsistent with their app names. To illustrate the defense to the type C attack, we develop a “repackaged” Twitter app. It is also detected by AccountFish when trying to send out the credentials (Figure 2.13(c)).
Table 2.2: Average Execution Time of the MobiFish Scheme

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Phases</th>
<th>Execution time (s)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OCR-based technique</td>
<td>Taking screenshot</td>
<td>0.015</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCR extraction</td>
<td>3.206</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SLD searching</td>
<td>0.094</td>
<td></td>
</tr>
<tr>
<td>Non-OCR technique</td>
<td>Timestamp comparison</td>
<td>0.018</td>
<td></td>
</tr>
</tbody>
</table>

2.6.4 Overhead Evaluation

We have validated the efficacy of our proposed three sub-schemes through the above experiments. Next, we evaluate the usability (performance) by measuring the execution time of the MobiFish scheme. There are two major techniques used in MobiFish: searching the SLD in the text extracted from the screenshot, and blocking the SMS and socket connections for a certain period of time. The SLD searching is based on OCR technique, which is considered much more time-consuming. Hence, we evaluate the delay overhead of OCR-based techniques and other non-OCR techniques separately. The results are presented in Table 2.2.

The OCR-based techniques can be roughly divided into three phases: taking a screenshot, extracting text from the screenshot, and searching if the SLD exists in the text. Since the SLD searching technique is applied in all the three sub-schemes, our testing samples include (1) the 13 official websites listed in Table 2.1 and 13 corresponding phishing web pages, and (2) 10 popular benign apps that support persistent account and the phishing apps developed by us. The average execution time of the three phases are 0.015, 3.206, and 0.094 seconds, respectively. As we can see, the OCR extraction phase holds 96.7% of the delay overhead (3.206 out of 3.315 seconds). Regarding the non-OCR techniques, we tested the SMS and socket connection blocking for a period of time. The app samples used are the same 10 popular benign apps and the phishing apps as above. The delay overhead is very low (0.018 seconds) because we only needs to decide whether the blocking period has expired, through a single comparison of two timestamps.
Note that the above phishing detection techniques are performed in parallel to the regular functions (e.g., web page and app authentication), hence the user experience will not be influenced. When the checking starts, a toast (a quick message) is displayed to notify the user. Users are suggested to submit the credentials after the checking is done. We believe that in most cases, the checking can be finished before a mobile user inputs the credentials. Meanwhile, we seek to improve the OCR technique so that the extraction process can be expedited.

2.7 Literature Review

2.7.1 Conventional Phishing Web Page Detection

Web users have been suffering from phishing attacks since their first appearance in 2003. Researchers have proposed many solutions (such as alert protection and phishing detection) to defend against phishing attacks.

Alert protection is a simple notification when a user is entering sensitive information. Kirda et al. proposed AntiPhish [38], which tracks the sensitive information of a user and generates warnings whenever the user attempts to give away this information to a website that is considered untrusted. However, this scheme cannot automatically check and detect phishing attacks. Instead, users have to judge by themselves after being warned.

In addition, many phishing detection tools have been designed for phishing on PC web pages. Based on the methods used, they can be generally categorized into two groups: heuristics schemes and blacklist schemes. Heuristics schemes outperform blacklist schemes since they can deal with new phishing sites without having to wait for an update. Usually, heuristics schemes for phishing detection utilize other techniques such as machine learning techniques [39–41] and search engine [41, 42].
CANTINA [42] is a content-based approach to detecting phishing websites, and it adopts TF-IDF information retrieval algorithms. Garera et al. [39] proposed a heuristics-based scheme which identifies several generic features of phishing URLs, and uses these features in a logistic regression classifier. CANTINA+ [41] is a comprehensive feature-based solution for web page phishing which combines machine learning and search engine techniques. However, existing heuristics used in phishing detection are all based on features extracted from the HTML source code. As we have shown in Section 2.3, HTML source code should not be trusted since it may not reflect the actual content presented to users.

Based on the assumption that the most spoofing phishing sites are those whose visual appearances look identical or very similar to authentic sites [43, 44], several similarity-based phishing detection approaches are proposed. SpoofGuard [45] uses URLs, images, links, and domain names to check the similarity between a given page and the pages previously stored. Afroz et al. proposed PhishZoo [46] that uses the profiles of trusted websites’ appearances built with fuzzy hashing techniques to detect phishing. PhishZoo makes profiles of sites that consist of fuzzy hashes of several common content elements (e.g., URL, images, most used texts, HTML codes, script files, etc.), which are related to the structure and appearance of the sites. They further enhanced their phishing detection scheme by adding displayed images into profiles and utilizing SIFT image-matching algorithm [34]. However, similarity-based approaches also depend on HTML source code and cannot detect phishing sites with different appearances.

GoldPhish [35] utilizes the OCR technique for phishing detection in PC browsers. OCR is used to extract text from images found in web pages (e.g., the company logo), then it is compared to the top-ranked domains from Google’s search service. However, OCR performance on PC is demonstrated to be limited in both speed and accuracy. Our lightweight scheme works with mobile browsers, and does not depend
on external search engines.

2.7.2 Mobile Phishing Detection

Mobile phishing attacks are emerging as a significant threat for mobile users. Niu et al. [30] discussed the weakness of mobile browsers caused by the hardware limitation of mobile devices. Felt et al. [32] examined the mobile phishing threats by detailing several phishing attack models during control transfers. Both works give some suggestions on phishing mitigation. Niu et al. [30] advised redesigning the browsers to make the origin and authenticity of the site more apparent to users. However, it is very difficult to add more features to the mobile user interface due to the limited screen size. And even if they are added, some web users will still ignore the identifier [47]. Felt et al. [32] proposed to add an always-present identity bar that displays the name of the current foreground application or the domain name of the current web page. Bianchi et al. [48] implemented an identity indicator for apps in the system navigation bar, in which the Extended-Validation (EV) HTTPS infrastructure is used to validate the app developers. Marforio et al. [49] applied personalized security indicators (an image chosen by the user that is displayed in the login UI) to mobile apps. However, all these indicator-based approaches require the user to make the final decision.

Another group of phishing defense techniques employ a unified and trusted login UI for apps. ScreenPass [50] provides a trusted software keyboard which allows users to specify their passwords’ domains (i.e., to tag passwords) together with the credentials. The OCR is used to ensure that passwords are entered only through the trusted software keyboard. This approach needs the user to switch to the secure keyboard when entering password, tag the password, and make the final decision, which may greatly degrade the user experience. VeriUI [51] utilizes an attested login which
augments user credentials with a certificate about the software and hardware that handle the credentials. However, this work requires not only the user effort, but also modifications to the client apps.

Moreover, a proxy service is designed in [30] which performs anti-phishing filtering against the URLs, page content, or user context. But it has to be downloaded and configured manually in the browser. Users also need to be able to authenticate the identity of the proxy (attackers can also set up fake proxies). Hou et al. [52] developed a defense scheme which loads hook into iOS so that the system interrupts the user when sensitive information is being entered into applications not in the whitelist, and prompts the user to decide whether to continue or not. However, this idea is quite similar to AntiPhish [38], which only gives a warning of credential rendering instead of phishing vulnerability. Cooley et al. proposed Trusted Activity Chains [53] to protect activities from spoofing preventions. However, it is the developer’s responsibility to annotate the chain of activities that should not be interrupted. This means that existing apps are not protected, and the developers may not assume the extra burden of annotation.

Our work differs from previous works in three folds: (1) MobiFish is a completely automated defense scheme, and users do not need to make the final decision. Although it is users who finally remove the phishing app, the user effort is trivial. Actually, they do not need to explicitly make the decision at all, since the only explanation for the login failure (with correct credentials) is a phishing attack. (2) No change is required to the browser/app/website’s UIs, as MobiFish is compatible with all existing websites and apps (no developer effort is needed). (3) The phishing attacks targeted at the persistent account is discovered and handled by the AccountFish scheme.

Besides, mobile phishing attacks could also be in the form of Emails or SMS. Phishing emails usually request users to click a link to a fake website where the
user is prompted to enter login credentials [54] [55]. The SMS phishing attacks (SMiShing) [56] [57] usually trick users into visiting a fraudulent website or calling a phishing number, where the victims are enticed into providing the credentials. The fraudulent websites could be defended by WebFish. But the detection of the phishing voice calls is beyond the scope of our work. Most voice phishing (Vishing) uses the VoIP technique in which the phone number is dynamically generated, we left this part for future work.

2.8 Chapter Summary

In this chapter, we study the important issue of mobile phishing detection. We propose MobiFish, a novel automated phishing defense scheme for mobile platforms. We identify the weaknesses of the heuristics-based anti-phishing schemes that highly rely on the HTML source code of web pages. MobiFish resolves this issue by using OCR, which can accurately extract text from the screenshot of the login interface so that the claimed identity can be verified. Compared to existing OCR-based anti-phishing schemes (designed for PC only), MobiFish is lightweight as it works without using external search engines or machine learning techniques. Besides, MobiFish can also detect the app phishing attacks and account phishing attacks. We implement MobiFish on a Google Nexus 4 smartphone running the Android 4.2 OS. Our evaluation demonstrates that MobiFish can effectively detect and defend against mobile phishing attacks.

Here are the main contributions of this chapter:

1. We find the weakness of previous heuristics-based security schemes for conventional web page phishing, and propose a lightweight detecting strategy that utilizes optical character recognition for web phishing and app phishing attacks.
2. We present account registry phishing attacks. To the best of our knowledge, we are the first to give detailed formulation and defense scheme for this type of attacks.

3. We propose MobiFish, a novel automated lightweight anti-phishing scheme for mobile phones. We implement MobiFish on a Google Nexus 4 smartphone (Android 4.2).

4. We evaluate the effectiveness and usability of MobiFish with phishing URLs and phishing apps. We also measure the delay overhead of MobiFish.
CHAPTER 3

MOBILE APPLICATION CLICKJACKING ATTACKS

Smartphones bring users lots of convenience by integrating a variety of useful functions people may need. While users are spending more time on their phones, have they ever questioned of being spoofed by the windows they are interacting with? This chapter conducts a thorough study of the mobile clickjacking attack. We first present how the clickjacking attack works and the key points in order to remain undiscovered. Then, we evaluate its potential threats by exploring the feasibility of launching clickjacking attacks on various UIs, including system app windows, third-party app windows, and other system UIs. Finally, we propose a system-level defense scheme against clickjacking attacks on Android platform, which requires no user or developer effort and is compatible with existing apps. The performance of the countermeasure is evaluated with extensive experiments. The results show that our scheme can effectively prevent clickjacking attacks with only a minor impact to the system.

3.1 Introduction

Smartphone continues its popularization worldwide and has become an important part of people’s daily lives. Android is the most popular and the best-selling smartphone OS, holding over 80% of global smartphone market share [58]. However, security and privacy issues are a widely recognized problem of Android, mainly
because it is open-source and attackers can find security vulnerabilities from the source code.

The security of UI is particularly important, since mobile users interact directly with the UIs of the system as well as third-party apps. Specifically, users receive most information visually from the UI, and give their inputs in terms of touch, click, and key entry to the UI as well. The manipulation of UIs can pose huge threats to the interaction between user and the phone. In general, the identities of two UIs should be treated carefully regarding the UI security: the UI that the user thinks of interacting with and the UI that is actually taking the user inputs. It is very important to guarantee that these two identities are consistent, otherwise it indicates that the user is spoofed and an app is receiving the user inputs while it is not supposed to. When talking about illegally gaining user input, most people immediately blame phishing attacks. However, the app to which user inputs are sent to is not necessarily the malicious one. Instead, it can be the victim in a clickjacking attack. For example, it is covered intentionally by a malicious UI that is “transparent” to user inputs (does not accept user inputs). As the result, the victim app is “forced” to take the user inputs while the user is expecting to interact with the app on top. Hence, we look into the mobile UI spoofing attacks in two categories based on whether the malicious UI intends to obtain the user input, namely the phishing attack (steal user input) and clickjacking attack (redirect user input to the victim UI).

In this chapter, we focus on mobile clickjacking attacks. We start with the general pattern and key steps to implement a clickjacking attack. Then, we give a detailed analysis of the potential risks posed by clickjacking. Finally, we propose an automatic, lightweight and effective defense scheme to defeat clickjacking attempts, which is able to overcome the limitations of all existing solutions. All different types of clickjacking attacks and the defense mechanism are implemented on a Nexus 4 smartphone running Android 5.0 system. The effectiveness and overheads of the proposed scheme are
evaluated with extensive experiments.

3.2 Background

3.2.1 Android UI Basics

There are two confusing concepts that we want to explain in the first place: view and window. In Android, a View object is the basic building block for UI components (e.g., buttons, images, text fields, etc.). Views can be used to draw graphical contents. A Window object is an abstract base class for a top-level window look and behavior policy. By comparison, View objects are closer to the conventional windows we considered to display content as graphical user interfaces (GUIs), while Window objects are more likely a framework to hold the Views. In this chapter, both terms “view” and “window” refer to the UI implemented using View object.

3.2.2 Overview

Clickjacking attack is also known as “UI redress attack”. It happens when a malicious app inserts an opaque layer (or in very low transparency) on top of the screen, to trick a user to click on a specific position. The click event seemingly going to the top front window actually goes to the target window underneath. If carefully designed, the user may trigger a concealed button or link in the underlying window. Note that the conventional webpage clickjacking attacks have been well studied in previous works [59–64], and most of them can be migrated to mobile platform to protect mobile browser and webpages. In our work, we focus on clickjacking attacks on mobile application UIs and system UIs. An illustration is presented in Figure 3.1, in which the attack window appears as a game menu and covers on top of the target window. When the user clicks to start the game, the click is instead received by the
“SEND” button in the target window (e.g., a premium SMS). Obviously, the target window is launched by the malicious app after the attack window has blocked user’s sight. This is because if instead the target window is launched first (by the user), then the only way that the clickjacking can succeed is to have the user open the malicious app (attack window) while the target window is on front. This condition (predicting user behavior) is too strong for any real-world app.

Therefore, the attack window needs to be added before the target window, and remains on top when the target window is launched; otherwise, the user will see the target window launched without command. Intuitively, the attack window cannot be in the same type as normal apps, which will be covered by the target window launched later. In fact, the attack window must be a floating window which has higher priority to stay on top of normal app windows (detailed in Section 3.4.1). Besides, the attack window has to be untouchable (does not take user inputs), so that user inputs can penetrate this camouflage layer and take effect on the target window below.

### 3.3 Literature Review

A recent work conducted by Bianchi et al. [48] comprehensively study the Android UI attacks. In their work, GUI confusion attacks, including both the phishing attack
and clickjacking attack, are handled together. They first develop a static analysis approach to detect apps that try to interfere with the UI in response to some action taken by the user (or another app). However, unlike the phishing attacks that are performed right after the user launches the target app login UI, a clickjacking attack does not have a specific timing (the launch of the attack window and the target window are both controlled by the malicious app). Hence, clickjacking attacks can bypass this method as they are independent of user’s or other app’s action. Meanwhile, some benign apps are captured instead (e.g., locker apps will always stay on top of the app that is being protected until the user authenticates herself). They also devise an on-device defense by modifying the Android system. A trusted indicator is added to show the identity of the app that the user is interacting with, and its developer. Extended-Validation (EV) HTTPS infrastructure is used to verify if an app is indeed associated with a specific domain name. Warnings will be given if the user is interacting with an unverified app or an external window is present over the top activity. The limitations of this solution are three-fold: (1) Users have to make the final decision, which requires mobile users to understand the potential attacks they are facing and evaluate the risk in each specific situation. (2) Yellow alert will be issued to a verified app covered with an external window. However, some apps create an “always visible” window (e.g., Facebook Messenger provides the ability to chat while using other apps), and false alarms are generated when dealing with such benign apps. (3) The HTTPS EV certificates adopted will force the apps to be associated with domain names, and the certificate itself costs more than $100 each year. The verification of apps and developers is not a straightforward approach to excluding benign apps. The reason why they are using this approach as a complement is that the common features of different mobile UI attacks alone are not sufficient to detect any of the UI attacks. Instead, we split the general UI attacks to phishing and clickjacking, and develop separate detection schemes based on each’s unique features.
Next, we discuss the existing solutions that are specific to the clickjacking attacks on mobile platform.

Niemietz et al. [65] study the mobile app clickjacking attacks through case studies. They propose to add a security layer between all neighboring apps (without implementation), so that no user input can pass across apps. However, some benign app windows are designed to be untouchable (transparent to user input) and floating on top of the screen (illustrated in Section 3.4.1). Blindly rejecting pass-through user inputs could cause serious incompatibilities with such apps. Besides, Android system creates flags like `FLAG_NOT_TOUCHABLE` and `FLAG_NOT_TOUCH_MODAL` for UIs, to offer the option to let the input events above/outside of a window to be delivered to the window below. The proposed solid security layer will spoil the intrinsic design of Android, and cause many existing apps to malfunction.

Android framework offers a touch-filtering mechanism as an option for app developers to protect the windows of their apps from receiving any user input event when obscured by another visible window. Specifically, the touch filtering can be activated by calling the `setFilterTouchesWhenObscured` function, or by setting the `android:filterTouchesWhenObscured` attribute of a layout to true (for all views contained or only selected views). This method is also suggested by Niemietz’s follow-up work [66]. When enabled, the protected views will simply discard inputs whenever a toast, dialog or other window (may belong to a benign app) appears above. However, this solution is incompatible with benign apps that contain a floating window. This explains why Android phone manufacturers will not set up this flag by default for their system apps. What we need is a smart defense scheme that can make the filtering decision in real time and only filter out misled input.

Fernandes et al. [67] find that the security indicator in [48] checks the identity of the foreground app periodically, so that the malicious app can launch clickjacking
attacks right between two consecutive checking points. The timing of the binder IPC calls can be leveraged as the side-channel to predict the next check. Instead, they propose Overlay Mutex which guarantees that a background non-system app cannot overlay a window on top of another app’s window while it is using soft keyboard. However, it will block the pop-up notification window of benign third-party apps, and is conflicting with Android flag FLAG_NOT_FOCUSBABLE which does allow an overlaid window to interact with the soft keyboard. We need to design the clickjacking detection scheme to be compatible with Android system and all existing apps.

3.4 Mobile Clickjacking Attacks

In this section, we describe the mobile clickjacking attacks in detail. Specifically, we first introduce the floating window and the target window, which are the basic components in a clickjacking attack. Next, we present a step-by-step analysis of the general attack model. Then, we discuss the potential threats posed by clickjacking attacks.

3.4.1 Floating Window

Floating window is a special type of window that can remain on top of normal app windows. By using a floating window, UIs of multiple apps can be displayed together, thus facilitating the cooperation between apps and simplifying user operations. For instance, some apps keep a small and semitransparent floating icon on screen, to provide quick access to useful functions (e.g., switching network connections, GPS, taking photos, etc.). These apps are welcomed by mobile users because of their convenience. These functions are usually accessible within 2 clicks, which otherwise would require tedious app switching operations including pausing the current activity, going to home menu or recent app list, activating the target app, and getting
back to the original app after completion. Some other apps create an untouchable semitransparent floating window. For example, in screen camera apps, a full-screen translucent camera preview is projected into a floating window so that pedestrian users can see the traffics/obstacles ahead while texting; system performance monitor apps can display the real-time CPU and memory usage in such windows.

Usually the floating window is created by a service, so that it can still be floating on screen even if the host app has been brought to the background. Besides, Android system enforces a “solid” layer to block the penetration of user inputs between windows created by different activities, and an untouchable floating window that wants the user inputs to pass through (like the screen camera apps) can bypass this restriction only if it is running in a service. A floating window can create fancy UIs and improve user experience. However, it may be used to launch clickjacking attacks if carefully designed by the attacker. In the rest of this chapter, the term “floating window” refers to the malicious attack window used to perform clickjacking attacks, unless explicitly specified.

Unlike in an activity where associated views are loaded together with the activity, views in a service have to be added explicitly. This can be achieved using the addview API of the WindowManager. When calling this API, several parameters can be specified in LayoutParams which will affect how the window is laid out, including the width/height, position, general window type, behavioral flags, and desired bitmap format. The size and position are common parameters. As an attack window, the floating window has to occupy the entire available area on screen. This is because only a full-screen floating window (except the status bar and the navigation bar) can ensure that the user does not get any visual clue of the attack happening underneath (i.e., the loading of the target window). The fifth parameter window format defines the desired bitmap format. Any of the formats defined in PixelFormat can be used for floating window. Below we explain the other two parameters: Type and Flags.
Table 3.1: Window Types for Floating Window

<table>
<thead>
<tr>
<th>Window Type</th>
<th>Layer Value</th>
<th>Permission Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type_Phone</td>
<td>3</td>
<td>System_ALERT_Window</td>
<td>user interaction with the phone (in particular incoming calls)</td>
</tr>
<tr>
<td>Type_Toast</td>
<td>7</td>
<td>None</td>
<td>transient notifications</td>
</tr>
<tr>
<td>Type_Priority_Phone</td>
<td>8</td>
<td>System_ALERT_Window</td>
<td>priority phone UI, needs to be displayed even if the keyguard is active</td>
</tr>
<tr>
<td>Type_System_Alt</td>
<td>10</td>
<td>System_ALERT_Window</td>
<td>system alert window, such as low power alert</td>
</tr>
<tr>
<td>Type_System_Overlay</td>
<td>19</td>
<td>System_ALERT_Window</td>
<td>system overlay windows</td>
</tr>
<tr>
<td>Type_System_Error</td>
<td>22</td>
<td>System_ALERT_Window</td>
<td>internal system error windows</td>
</tr>
</tbody>
</table>

3.4.1.1 Window Type

To achieve the goal of floating, we pick out the window types that are prioritized to stay above normal app windows, and permitted to be used in a third-party app (the malicious app), as listed in Table 3.1.

In Android, each window is associated with a Z value (Z-order). Windows with a larger Z-order are placed on top of those with a lower Z-order. The Z-order of a window is determined by both the window type (Layer Value) and its position in the “window stack”. The window management is a complicated task: When there is a change in the “window stack”, Z-orders of all windows will need to be re-calculated. However, in terms of detecting a malicious floating window, we only have to know two basic rules: (1) For windows with different Layer Values, the one with a larger Layer Value can stay above the lower ones. (2) For windows with the same Layer Value, the most recently launched one will appear on top of the previous ones. In a clickjacking attack, the floating window is launched in advance and needs to cover the target window launched afterwards, so the floating window must have a Layer Value greater than the target window.

In Table 3.1, the window types are listed in an increasing order of the Layer Value. For a given target window, the attacker only has to pick one window type with a
<table>
<thead>
<tr>
<th>Window Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLAG_NOT_TOUCHABLE</td>
<td>this window can never receive touch events.</td>
</tr>
<tr>
<td>FLAG_NOT_FOCUSABLE</td>
<td>this window does not need to interact with a soft input method.</td>
</tr>
<tr>
<td>FLAG_ALT_FOCUSABLE_IM</td>
<td>invert the state of FLAG_NOT_FOCUSABLE with respect to how this window interacts with the current method.</td>
</tr>
</tbody>
</table>

greater Layer Value. Meanwhile, we noticed that all of the listed window types require SYSTEM_ALERT_WINDOW permission except TYPE_TOAST. According to a study of 1260 malware samples and 1260 top free benign apps [68], SYSTEM_ALERT_WINDOW permission is not among the top 20 commonly requested permissions of both malware and benign app samples. And it is widely used among some categories of apps including anti-virus apps, locker apps, etc. Still, if the malicious app can avoid using it, the malware could become less suspicious. Note that if a TYPE_TOAST floating window is created with addView API, it will not go disappeared after a certain period of time like Toast.show() API does. Instead, the floating window remains on the screen until removeView API is called to explicitly remove the window.

3.4.1.2 Window Flag

Flags provide various behavioral options for windows. Unlike the single-choice window type, there can be multiple flags to be set for a given window. We describe the flags relevant to floating windows in Table 3.2.

Since the floating window has to be untouchable, the FLAG_NOT_TOUCHABLE flag must be set so that all click and touch events will be passed down to the target window. The FLAG_NOT_FOCUSABLE flag controls whether a given window can take key input focus and other button events (e.g., back button). If set, these will be delivered to the next focusable window behind. In Android, the input focus can only be given to one window, namely the top focusable window. Hence if the
FLAG_NOT_FOCUSABLE flag is not set for the floating window, the target window will become unfocusable. The FLAG_ALT_FOCUSABLE_IM flag is to invert the state of FLAG_NOT_FOCUSABLE. If both flags are set, the window will behave as if it needs to interact with the input method; while if FLAG_NOT_FOCUSABLE is not set and this flag is set, then the window will behave as if it does not need to interact with the input method.

Generally, the attacker wants to trick the user to operate on the target window with a few clicks. Attacks with a large number of clicks/touches or text input are much harder to accomplish, as the attacker will have to design a series of UIs and ensure that the user will always click on the desired position. The floating window is advised not to set any one of the FLAG_NOT_FOCUSABLE and FLAG_ALT_FOCUSABLE_IM, to keep the input focus from the target window. In this way, even if the user clicks on the text field of the target window (if there is any) by accident, the input method will not be activated and give sound/vibration feedbacks to alert the user. A special case is that FLAG_NOT_TOUCHABLE and FLAG_NOT_FOCUSABLE will be automatically added for windows of TYPE_SYSTEM_OVERLAY if they are absent (Android thinks that such types of windows must not take input focus, or they will interfere with the keyguard).

3.4.2 Target Window

The target window can belong to a third-party app, system app or other system UIs (e.g., confirmation dialogs). Some of these UIs’ functionalities are security-sensitive and require specific permissions to use programmatically, e.g., SMS messenger, camera, system settings, etc. However, the malicious app could launch the target app UI while the floating window is on, and entice the user to manually operate according to the attacker’s will (e.g., pressing attractive dummy buttons which sit
right on top of actual buttons). As the invocation of the target UI usually does not need any permission, the clickjacking attacks can “achieve” the malicious intention without declaring those sensitive permissions.

The launching of the target UI can be done either programmatically or manually. Manual launch could take lots of extra efforts (e.g., navigating in the app or settings menu). By comparison, the code-based approach will not increase the overall attack complexity, and is much stealthier. The invocation of the target UI through code requires inter-application communication. Android provides two types of channels for apps to communicate with each other: Intents and Schemes. Furthermore, there are two forms of intents: explicit intents and implicit intents. Explicit intents specify the exported app component to start by class name, while implicit intents declare a general action to match against the intent filters of all exported components (i.e., components that are invokable by other apps). Scheme is implemented based on intent, which allows an app or website to use a URL to invoke another app that has registered the scheme of that URL.

A practical issue for implicit intents and schemes is that when multiple compatible intent filters (from different components) or scheme “handlers” are available, a system dialog is displayed which lists all candidate apps alphabetically and prompts the user to manually select one. Though the malicious app can calculate the target app’s position by scanning all components via PackageManager, a simple bypass for this is to directly use an explicit intent. We use explicit intent to launch target windows in our experiment.

3.4.3 Clickjacking Attack Implementation

So far, we have introduced the fundamental knowledge of the floating window and target window. Next, we give a step-by-step analysis on how to implement a
clickjacking attack.

3.4.3.1 Step 1 - Launch the Floating Window

Through former study, we have determined that the first step to perform a clickjacking attack is to launch the floating window. That is, when the malicious app starts to attack, it first adds a full-screen floating window on top of the screen. The type of floating window can be chosen from Table 3.1, whose Layer Value has to be greater than the pre-selected target window. The flags of the floating window must contain FLAG\_NOT\_TOUCHABLE, while it is highly suggested that FLAG\_NOT\_FOCUSABLE and FLAG\_ALT\_FOCUSABLE\_IM are not set.

3.4.3.2 Step 2 - Launch the Target Window

The target window is launched right after the floating window, to receive the user inputs. Blinded by the floating window, the user would “click” the dummy object while the click event is actually delivered to the target window. One may consider that as an end for clickjacking attacks. However, such attacks may be discovered immediately after a one-time success. Since the floating window is designed to pass over the user inputs, the malicious app may not be notified of when the input events take place. This is a crucial issue for clickjacking attacks as the user will sense the abnormality if the malicious app cannot “react” in a timely manner to her input action. In order to remain undiscovered, the malicious app must either “listen” to the input taking effect on the target window, or find a way to detect the expected result of the user input.
3.4.3.3 Step 3 - Monitor User Input and Respond

The durability of the malicious app is also an important factor from the attacker’s point of view, which has been neglected in all previous works. After the user falls into the clickjacking trap (e.g., click onto the fake UI), the malicious app needs to know as soon as possible when the user input is performed, and react in a way as if the user is really interacting with it. In our work, these “after-attack” disguises are considered. Based on each particular target window, there could be three kinds of ways for the malicious app to track the user input events:

- As the caller of the target window, the target app may issue a callback to the malicious app (e.g., calling the target activity with `startActivityForResult()` and register the `onActivityResult()` callback method).

- The target app (or the system) may issue a broadcast intent to inform all related modules about the specific user action or the phone’s state change.

- The malicious app may detect if the expected user action has been made by observing the relevant database.

If it is the first case, the malicious app can directly insert the code for proper response into the callback function. For example, the malicious app wants to take a picture by stealthily starting the camera app with `ACTION_IMAGE_CAPTURE` intent. After the user is spoofed to click the capture button, the image is returned to the malicious app. Then the malicious app can save or send out the image while on the surface (floating window) starting a game as the user has expected. Note that for camera-based attacks, the phone needs be set to the silent ringer mode `RINGER_MODE_SILENT`, in which sound and vibration will be shut down. The audio volume and vibration state will be recovered (if modified) after the attack.

If the target app does not give a callback, the malicious app will have to seek for other channels. In Android, some of the user actions will be broadcasted so that the
relevant apps can keep up with the most recent phone status and adapt themselves accordingly. For example, the malicious app may launch the date settings panel by sending an intent with DATE_SETTINGS action, to trick the user to modify the date (and/or time) so that the scheduled calendar events, reminders, and alarming events are all disturbed. Because the time & date setting is considered security sensitive, the required permission SET_TIME is not granted to third-party apps. However, this protection strategy can be easily bypassed via a clickjacking attack. Meanwhile, the malicious app can register a broadcast receiver with the TIME_SET action to catch the moment that the date/time is changed.

Under other situations where no broadcast channel is available to listen to, the malicious app could register an “observer” for the relevant database as a way to eavesdrop on the user action of interest. For instance, there is a broadcast intent SMS_RECEIVED_ACTION for incoming SMS but no notification is issued for outgoing SMS. However, Android uses content providers to store common data such as contact information, calendar information, media files, messages, call logs, etc. After programmatically filling the phone number and content into an SMS draft, the malicious app can monitor the SMS database (basically a content provider that contains all SMS messages). Specifically, a ContentObserver is registered with “content://sms” as the content URL (no SMS-related permission is needed). Since the time that the user takes to click and send the message is very short (we assume the user is spoofed and tempted to click without hesitation), the change in SMS database during that period is sufficient to indicate that the expected “click to send” action by the user has been made.

In either of the three circumstances above, after the user input is detected (the clickjacking attack has succeeded), the malicious app needs to respond instantly and properly, according to the dummy function that the user selected from the floating window. Based on whether to continue the attack, the floating window may be
removed (the attack ends) or another target window may be loaded (the attack continues).

3.4.4 Potential Risks of Clickjacking Attacks

The threats posed by clickjacking are decided by the various target windows under attack. In general, the ideal target should be invokable through code, which means the component holding the target window (usually an activity) has to be exported (i.e., the \texttt{android:exported} attribute is set to true). Additionally, the malicious app should be able to monitor the user action on the target window. To evaluate the potential threats, we present the typical clickjacking attacks targeted on system apps, third-party apps and system UIs, respectively.

3.4.4.1 Clickjacking Attacks on System Applications

We first summarize the commonly used public components of system apps that are vulnerable to clickjacking attacks (listed in Table 3.3). The second column shows the component name that can be used to explicitly invoke the target window. The after-attack monitor channel is presented in the fourth column. As we have found, the target windows of all victim components are of the same type \texttt{TYPE\_BASE\_APPLICATION}, which is the normal window type. All these target windows have the same Layer Value of 2, which means that they will be shielded by any of the floating window types listed in Table 3.1. An exception is that sometimes user operations may trigger toast messages (\texttt{TYPE\_TOAST}). For instance, after the user modifies a contact, a toast message “Contact saved” will be displayed. When launching attacks associated with toast messages, the floating window type must have a Layer Value greater than \texttt{TYPE\_TOAST}.

The target windows could belong to media apps (camera, sound recorder),
<table>
<thead>
<tr>
<th>Target app</th>
<th>Target Component</th>
<th>Intent Action (Scheme)</th>
<th>Monitor Channel</th>
<th>Permission Bypassed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camera</td>
<td>CameraActivity</td>
<td>IMAGE_CAPTURE</td>
<td>Callback</td>
<td>CAMERA</td>
</tr>
<tr>
<td>Camera</td>
<td>VideoCamera (alias)</td>
<td>VIDEO_CAPTURE</td>
<td>Callback</td>
<td>CAMERA</td>
</tr>
<tr>
<td>Contact</td>
<td>ContactSelectionActivity</td>
<td>PICK</td>
<td>Callback</td>
<td>READ_CONTACTS</td>
</tr>
<tr>
<td>Contact</td>
<td>ContactEditorActivity</td>
<td>EDIT</td>
<td>Listen to contact database</td>
<td>WRITE_CONTACTS</td>
</tr>
<tr>
<td>Dialer</td>
<td>DialtactsActivity</td>
<td>DIAL</td>
<td>Listen to contact database</td>
<td>CALL_PHONE</td>
</tr>
<tr>
<td>Documents</td>
<td>DocumentsActivity</td>
<td>GET_CONTENT</td>
<td>Callback</td>
<td>READ_EXTERNAL_STORAGE</td>
</tr>
<tr>
<td>Package installer</td>
<td>PackageInstallerActivity</td>
<td>INSTALL_PACKAGE</td>
<td>Callback</td>
<td>INSTALL_PACKAGES (system only)</td>
</tr>
<tr>
<td></td>
<td>UninstallerActivity</td>
<td>UNINSTALL_PACKAGE</td>
<td>Callback</td>
<td>DELETE_PACKAGES (system only)</td>
</tr>
<tr>
<td>Settings</td>
<td>WirelessSettingsActivity</td>
<td>WIRELESS_SETTINGS</td>
<td>Broadcast receiver</td>
<td>WRITE_SETTINGS</td>
</tr>
<tr>
<td></td>
<td>WifiSettingsActivity</td>
<td>WIFI_SETTINGS</td>
<td>Listen to settings database</td>
<td>CHANGE_WIFI_STATE</td>
</tr>
<tr>
<td></td>
<td>DateTimeSettingsActivity</td>
<td>DATE_SETTINGS</td>
<td>Broadcast receiver</td>
<td>SET_TIME (system only)</td>
</tr>
<tr>
<td>Sound recorder</td>
<td>SoundRecorder</td>
<td>RECORD_SOUND</td>
<td>Callback</td>
<td>RECORD_AUDIO</td>
</tr>
<tr>
<td>Messaging</td>
<td>ComposeMessageActivity</td>
<td>SENDTO</td>
<td>Listen to SMS/MMS database</td>
<td>SEND_SMS</td>
</tr>
</tbody>
</table>

Table 3.3: Clickjacking Attacks on System Apps
telephony apps (dialer, messaging, contact), the package installer, the settings or the file explorer. By tricking the user to click on these windows, the attacker is able to perform security sensitive operations behind the user’s back. These operations are normally guarded by specific permissions (listed in the “Permission Bypassed” column of Table 3.3), some are not available for third-party apps (INSTALL_PACKAGE, DELETE_PACKAGES and SET_TIME). From this perspective, the clickjacking attack actually gives the malicious app a permission escalation [69] [70], by having the privileged system apps delegate the sensitive operations (performed by the user). The system permissions usually protect APIs that are extremely security sensitive. For example, INSTALL_PACKAGE permission allows an app to silently install other apps. If a malicious app acquires this capability (even if indirectly via clickjacking), it can install other malwares and cause more serious consequences.

3.4.4.2 Clickjacking Attacks on 3rd-party Applications

We also look into the third-party apps and check if their APIs could be targeted by clickjacking attacks. It has been reported in [71] that many third-party apps unintentionally expose their components that may be utilized by attackers. In our work, we mainly consider apps that explicitly share their APIs, e.g., apps that release their own software development kits (SDKs) for other apps to use their services. This is quite popular especially among social networking apps which allow users to quickly spread information via their social networks. We use Twitter and Facebook as examples to check their potential vulnerability to the clickjacking attacks.

The Twitter and Facebook SDKs provide a set of APIs to perform commonly used operations like tweet, update mood, share photos/links, load contents from Twitter/Facebook, etc. We implement the basic “tweet” and “share link” functions on Twitter and Facebook, respectively, to demonstrate the feasibility of launching
clickjacking on these apps. We suppose that the user has already logged into the client app, so that the malicious app can be authorized via a couple of clicks.

In Twitter, a TwitterLoginButton is placed in the target window (also launched by the malicious app). When the user is tricked to click the TwitterLoginButton, he/she will be brought to the Twitter authorization window (shown in Figure 3.2(a)), where the user is then tricked to click the “allow” button so that the malicious app is authorized to utilize Twitter APIs (e.g., perform a post). The post operation can be conducted programmatically as soon as it is authorized. Specifically, the malicious app calls the get_statusesService() method to get a StatusesService object for tweet post requests. Then a message is composed, and posted by calling the update() method of the StatusesService object.

The general procedure of attacking the Facebook client app is similar to Twitter. But the click on the login button is not necessary, since the actual method of logging in is private with the Twitter SDK, whereas with the Facebook SDK it is public. Hence, the malicious app can directly bring the user to the Facebook login authorization window (shown in Figure 3.2(b)) to gain user’s authorization.
via clickjacking. The posting action on Facebook, however, requires explicit user approval. After these two authorizations, the malicious app will have the privilege to post. For example, a GraphRequest object can be created to post a shared link to the user’s wall ("me/feed").

In summary, the unauthorized postings (the user is not aware of) on Twitter and Facebook both require two clicks, which is easy to achieve by clickjacking attacks. The authorization window will also give a callback to the caller app, so the malicious app can know the user input event instantly and react accordingly. Note that the explicit authorization is required only once (in the first time the malicious app tries to connect with the client app). After that, even if the malicious app has been closed, it can still directly perform operations through APIs.

3.4.4.3 Clickjacking Attacks on System UIs

We find that the Android system UIs are vulnerable to clickjacking attacks as well. A particular example is the screen recording API: starting from Android 5.0 Lollipop, a new “android.media.projection” API is created to facilitate the screen capturing and screen sharing, without the need of connecting the device to PC over the Android Debug Bridge (ADB) as required by the previous Android 4.4 KitKat. Specifically, the createVirtualDisplay() method of the “MediaProjection” class starts a screen capture session to capture the contents of the main screen into a Surface object. The invocation of this API does not require any permission, but a confirmation dialog of TYPE_BASE_APPLICATION will pop up asking for user approval to start. In a clickjacking attack, the user may be spoofed to click right on the “start now” button without realizing that the screen recording is being activated. Unlike attacking system apps (e.g., calling the camera to shoot a video), the ending of the screen recording does not need user operation (manually click to stop). The malicious app can just
set a timer (in a service) to release the VirtualDisplay after a certain amount of time. Hence, the whole attack process can be very simple and stealthy. By default, the recorded video is sent to a Surface object that can be displayed instantly. If the attacker wants to obtain the video, it can be saved as a video file using MediaRecorder. An alternative way is to extract images from the buffer by setting an ImageReader object as the callback of createVirtualDisplay(), in a frequency high enough to observe all user actions.

Screen recording is considered as a sensitive API, as all user actions will be logged. Apart from the violation of user privacy, an even more serious consequence is the leak of credentials (e.g., user name, password, credit card details) as the screen recording will capture the login and transaction processes as well. Usually, the account name entered stays unmasked while the password will turn into asterisks or bullets after a short moment of time (e.g., half second), which is long enough for the attacker to recognize each letter/number.

### 3.5 Countermeasure

The design of our defense scheme to clickjacking attacks is based on the features summarized in Section 3.4. To overcome the limitations of existing solutions, we add an independent module to the Android system for automatic detection of clickjacking attacks. With our system-level and real-time protection, users are relieved from checking a security indicator whenever a window pops up and developers no longer need to worry about the incompatibility side-effect caused by touch filtering. Besides, the defense scheme is much more flexible than enforcing a solid security layer between each two neighboring apps, benign pass-through inputs will not be blocked.

We start with the two essential windows in clickjacking attacks. The camouflage window must be floating, untouchable and preferably focusable. Another factor that
has never been considered in previous works is the transparency (controlled by the alpha attribute of window layout parameters). A floating window in a benign app is usually auxiliary for the user or other apps, hence is set to semitransparent (e.g. screen camera apps); while in clickjacking, it is used to cover the target window and has to be opaque or slightly transparent. By default, we can set an empirical threshold value 0.95 for alpha in the detection, considering that the user can barely see through the window with an alpha value as high as 0.95. Users can modify this threshold according to their preferences. In contrast, there is no common feature for target windows since the target can be any normal-type app window or system UI. Hence, the identity of the target window is not clear until a clickjacking attack is launched.

Next, we analyze the dynamic features of clickjacking. An important pattern is that a successful clicking attack will always end with a user click. This means the detection is not necessary to be performed every time a window shows up, but instead, only when an input event is being received. Note that here we mean the input has just been received by the operating system and has not been delivered to the target app. The detection logic is to presume the window that has received the input as the target window, then check if it is covered by any malicious floating window of another app. There could be multiple windows above, all of them need to be checked since we are not sure which one is actually displayed to the user (fake floating windows with full transparent view or minimized size may exist on top to “fool” the detector). If any of these windows matches the features of a malicious floating window, it is considered as a clickjacking attack.

To check if there is a floating window among windows above, the detector should keep a record of all visible windows. Since the “window stack” in WindowManagerService is not maintained in the Z-order, a full re-calculation of the Z-order of all windows is required even if we are only interested in (malicious) floating windows. To avoid the unnecessary workloads, we create an
independent clickjacking detection service (CDS) in parallel with other system services (e.g., ActivityManagerService, WindowManagerService, etc). Meanwhile, four probes are set up to catch window actions. As depicted in Figure 3.3, the four probes are hooked to the `addView()` and `removeView()` methods of the WindowManagerGlobal class, `dispatchAppVisibility()` method of the ViewRootImpl class, and the `binderDied()` method defined in the WindowState class. The `addView()` and `removeView()` methods are invoked when views are created/destroyed along with their hosting activity, or added/removed explicitly through WindowManager APIs. The `dispatchAppVisibility()` method is called when the hosting activity becomes visible/invisible (hence so as the associated views). A special situation is that an app is uninstalled when its view is being displayed. In this case, the `binderDied()` callback method of the view is invoked and a corresponding removal will be performed at WindowManagerService. By hooking these four methods, CDS is able to collect all window actions including “add/remove” and “turn visible/invisible”. Specifically, the creation and turn-to-visible actions go to the `CDS_addView()` API while the removal and turn-to-invisible actions go to the `CDS_removeView()` API.

In Android, each time a view is added through `addView()` API, a ViewRootImpl object is created in the app process along with the window parameters and an InputChannel object (used to deliver input events to the destined window). Then these information are sent to WindowManagerService so that a corresponding
WindowState object is created. Unlike the window management service WindowManagerService, the CDS is specified to detect malicious floating windows for clickjacking. Obviously, it only needs to keep the information related to floating detection rather than the whole WindowState objects. Hence, we define a new class WindowCDS to record windows in a floating-specific representation. Specifically, all floating-related window attributes are kept as instance variables including window type, window flags, and alpha. In addition, the package name is needed to determine if the “floating window” and the “target window” actually belong to the same app (not considered as an attack). Furthermore, a unique ID is required to identify windows in CDS. However, Android does not provide a unified ID for windows/views. We resolve this issue by extracting the number contained in the InputChannel object’s name as the window ID (all windows have different InputChannels). Note that even if a window is untouchable/unfocusable, it is still assigned with an InputChannel.

The WindowCDS objects are stored in the CDS window list according to the Z-order. When a new window is inserted, it will be placed according to the Layer Value of its window type (defined in the PhoneWindowManager class). If there exists other windows with the same Layer Value, the new one will be placed ahead of the old ones. The removal of a window is simple: just search the ID in the window list, and remove the one with the same ID; however, it may happen that the window to be removed is not found (already removed). This is because sometimes the window is turned invisible before it is removed, which means the CDS.removeView() API will be called twice. Hence, “window not found” is considered acceptable in CDS, no exception will be given.

So far we are able to maintain the windows information in CDS. To catch the input events, we add a hook to the onInputEvent() callback function of the WindowInputEventReceiver. In each ViewRootImpl object, a WindowInputEventReceiver is registered along with the creation of the InputChannel,
to receive input events from the low level input dispatching module. In `onInputEvent()` function, the inputs are added into the input event queue and finally delivered to respective apps. Our defense strategy is to perform the checking for clickjacking attacks before the input event is handed over to the client app. Specifically, upon receiving an input event, the ID of the receiver window and the base package name are sent to the CDS, to check if it is under clickjacking attack. The input will stay suspended until the checking is cleared. In CDS, a malicious floating window is confirmed with four conditions: (1) its host app is different from the receiver window’s host app; (2) its Layer Value is greater than the receiver window; (3) its flags contains `FLAG_NOT_TOUCHABLE`; (4) its `alpha` value is greater than 0.95. If no such window is found, it means no clickjacking risk is detected and the input event can be released to the receiver app. Otherwise, the user will be alerted.

### 3.6 Evaluation

In this section, we evaluate the performance of the proposed defense scheme through extensive experiments. Although currently no real-world malicious clickjacking app is available on market, we believe this is a practical and potentially dangerous problem like previous works [48] [65] [66] [67]. We implement our clickjacking defense mechanism on a Nexus 4 smartphone running Android 5.0 system, and we add a switch button in the settings menu (as shown in Figure 3.4) to
enable/disable the CDS (by default it is enabled).

3.6.1 Effectiveness

Our defense scheme is able to detect clickjacking attacks on all different kinds of target windows considered above. Figure 3.5 shows the UI of a malicious game app “Space Battle” we developed for illustration. This UI is actually a floating window, and the user input will be delivered to the target window below which is the confirmation window of screen recording. With the CDS enabled, as the user clicks the play button, an alert window will be issued to warn the user about the clickjacking attack detected. The alert window is of type TYPE_SYSTEM_ERROR, hence it will not be shielded by the floating window. Meanwhile, sound and vibration alerts are also made to catch the user’s attention. The name of the suspected app is displayed in the title. The user can directly uninstall the app by clicking “OK” button in the alert window. Note that our defense scheme can deterministically decide if an app is launching a clickjacking attack. This warning dialog is not requesting user’s own judgement. Besides, we also test with common benign apps, including the locker
apps, apps with a small “always visible” icon, and screen camera apps that utilize floating windows. All of them are correctly recognized as benign by CDS, and no false alarm is generated.

### 3.6.2 Overhead

CDS is running as an independent service which manages the floating-related information for windows, and provides the checking of clickjacking attacks for each user input event. As a much simplified version of `WindowManagerService`, the collection of floating-related information in CDS includes only five variables (window type, window flags, alpha value, package name, and window ID). The overhead for collecting these variables is too trivial that our experiments cannot give a convincing measurement, so we only present the overhead of the checking procedure. Figure 3.6 shows the average execution time of the checking process in CDS, the checking is repeated for 10 times for each target window: on UI of system apps, it takes 5.2 to 10.5 ms; on UI of 3rd-party apps like Facebook and Twitter, it takes 14.1 and 12.2 ms respectively; on system UI like screen recording dialog, it takes 16.7 ms. The time delay caused by clickjacking detection is considered acceptable.
Figure 3.7: Overhead of the Checking Procedure
Next, we measure the real-time CPU load, memory usage and energy consumption per click event when clickjacking detection is enabled/disabled. The CPU load is measured by a system monitor app Tinycore, the memory usage is obtained via the `getMemoryInfo` API, while the energy consumption is measured using the power profiling app Trepn Profiler. All three metrics are measured for 50 trials (groups). Each CPU and memory trial is the average of 90 readings, while each power consumption trial is the average of 10 readings in a one-minute profiling. As depicted in Figure 3.7, the 50 groups of measurements are drawn by the increasing order of the regular overhead (without CDS). The solid part on top of the bar represents the extra overhead brought by CDS. Our results indicate that CDS causes a mean increase of 0.84% in CPU load, 0.145% in memory usage, and 3.21% in power consumption (relative increment). The standard deviation over 50 trails is 0.239%, 0.076%, and 1.487%, respectively. Overall, our clickjacking detection scheme is quite lightweight, particularly regarding its CPU and memory overhead. Note that currently there is no accurate and direct measurement of energy consumption available for phones, our result on power consumption is an approximate reflection of CDS’s impact.

3.7 Chapter Summary

In this chapter, we conduct a comprehensive study on mobile clickjacking attacks. We present the key steps to implement a stealthy clickjacking attack and explored its potential threats. We first analyze the two major components of a clickjacking attack, namely the floating window and target window. Then we describe the key steps to implement a clickjacking attack. We also study how the malicious app can respond instantly after a successful attack to remain undiscovered, which has been neglected by previous works. Finally, an automatic countermeasure for clickjacking attacks is implemented and evaluated. The experimental results show that our defense scheme
The main contributions of this chapter include:

1. We systematically study mobile clickjacking attacks from the perspective of floating window and target window, separately. We discover and analyze more unique features of clickjacking including the window flags, transparency, etc., which make our detection scheme more accurate than existing solutions which mistakenly accuse some benign apps due to their coarse policies.

2. We investigate the “after-attack” disguises to keep the clickjacking undiscovered after one successful attack, which has not been considered in previous works. Specifically, we present three types of side-channels that allow the malicious app to listen to the user input events (different to [72], not necessarily lead to UI state change).

3. We explore a variety of clickjacking attacks, targeted on system apps, 3rd-party apps, and other particular system UI. The threat of clickjacking is better evaluated than previous works which only have a couple of examples for illustration.

4. Our detection scheme outperforms existing solutions, and it requires no user/developer involvement and is compatible with Android system design as well as existing apps. We implement the proposed scheme on real smartphone. The experimental results show that it is effective and efficient.
CHAPTER 4

MOBILE CAMERA-BASED ATTACKS

Today’s mobile smartphones are very powerful and many smartphone applications use wireless multimedia communications. Mobile phone security has become an important aspect of security issues in wireless multimedia communications. As the most popular mobile operating system, Android security has been extensively studied by researchers. However, few works have studied mobile phone multimedia security. In this chapter, we focus on security issues related with mobile phone cameras. Specifically, we discover and present several new attacks that are based on the use of phone cameras. We implement the attacks on real phones and demonstrate the feasibility and effectiveness of the attacks. Furthermore, we propose a lightweight defense scheme that can effectively detect these attacks.

4.1 Introduction

Although Android permission mechanism allows users to view all the permissions requested by a specific app, it is difficult for users to check an app simply based on the permissions commonly requested by both benign and malicious apps, since the malware may appear to be a benign app while using the permissions for malicious activities stealthily. Meanwhile, increasing number of apps specified to enhance security and protect user privacy have appeared on Android Application Markets. Most large anti-virus software companies have published their Android-version security apps, and tried to provide a shield for smartphones by detecting and blocking malicious apps. In addition, there are data protection apps that provide
user the capability to encrypt, decrypt, sign, and verify signature for private texts, emails, and files. However, mobile malwares and privacy leakage remain a big threat for mobile phone security and privacy.

Generally, when talking about privacy protection, most smartphone users pay attention to the safety of SMS, email, contact list, calling history, location information and private files. They may be surprised that the phone camera could become a traitor. For example, attackers could stealthily take pictures and record videos by using the phone camera. Nowadays, various types of camera-based applications have appeared in Android Application Markets (e.g., photography, barcode reader, social networking). Spy camera apps have also become quite popular. As for Google Play, there are nearly one hundred spy camera apps, which allow phone users to take pictures or record videos of other people without their permissions. However, believe it or not, phone users themselves could also become the victim. Attackers can implement spy camera in malicious apps such that the phone camera is launched automatically without device owner’s notice and the captured photos and videos are sent out to the remote attackers. Even worse, according to a survey on Android malware analysis [73], camera permission ranks at the 12th of the most commonly requested permissions among benign apps while it is out of the top 20 in malwares. The popularity of camera usage in benign apps and relatively less usage in malwares lower user’s alert to camera based multimedia application attacks.

Nowadays, people carry their phones everywhere, hence their phones see lots of private information. If the phone camera is exploited by a malicious spy camera app, it may cause serious security and privacy problems. For example, the phone camera may record user’s daily activities and conversations, and then send these out via the Internet or Multimedia Messaging Service (MMS). Secret photography is not only immoral but also illegal in some countries due to the invasion of privacy. Nevertheless, a phone camera could also provide some benefits if it is controlled well
by the device owner. For example, when the owner wants to check if someone has used his/her phone without permission, the phone camera could be used to record the face of the user. Besides, it can also help the owner find a lost phone.

In this chapter, we first conduct a survey on the threats and benefits of spy camera. Then we present the basic attack model and two camera-based attacks: the remotely-controlled real-time monitoring attack and the passcode inference attack. We run these attacks along with popular antivirus software to test their stealthiness, and conduct experiments on video-based passcode inference attack. The results demonstrate the feasibility and effectiveness of these attacks. Finally, we propose a lightweight defense scheme.

### 4.2 Literature Review

A number of recent works have studied the issue of obtaining private information on smartphones using multimedia devices such as microphone and camera. For example, Soundcomber [74] is a stealthy Trojan that can sense the context of its audible surroundings to target and extract high-value data such as credit card and PIN numbers. Stealthy audio recording is easier to realize since it does not need to hide the camera preview. Xu et al. [75] present a data collection technique using video camera embedded in Windows Phones. Their malware (installed as a Trojan) secretly records video and transmits data using either email or MMS. Windows Phones offer a function `ShowWindow(hWnd, SW_HIDE)`, which can hide an app window on the phone screen. However, it is much more complicated to hide camera preview window in Android system. In this work, we are able to hide the whole camera app in Android. Moreover, we implement advanced forms of attacks such as remotely-controlled and real-time monitoring attacks. We also utilize computer vision techniques to analyze recorded videos and infer passcode from user’s eye movement.
Several video based attacks targeting at keystroke have been proposed. The attacks can obtain user input on touch screen smartphones. Maggi et al. [76] implement an automatic shoulder surfing attack against modern touch-enabled smartphones. The attacker deploys a video camera that can record the target screen while the victim is entering text. Then user input can be reconstructed solely based on the keystroke feedback displayed on the screen. However, this attack requires additional camera device and issues like how to place the camera near victim without catching alert must be carefully considered. Moreover, it works only when visual feedback such as magnified keys are available. iSpy [77], proposed by Raguram, shows how screen reflections may be used for reconstruction of text typed on smartphone’s virtual keyboard. Similarly, this attack also needs extra device to capture the reflections, and visual key press confirmation mechanism must be enabled on target phone. In contrast, our camera-based attacks work without any support from other devices.

4.3 Threats and Benefits of Spy Camera

As mentioned above, the role spy camera plays depends on the way it is used and who is in control of it. In the following, we discuss some threats and benefits of using spy camera.

4.3.1 Leaking Private Information

Spy camera works as a thief if it steals private information from the phone. Firstly, the malware finds a way to infect the victim’s smartphone. For example, it appears to be a normal app with legitimate use of camera and Internet. On one hand, it performs the function it claims. While on the other hand, it runs background service to secretly take pictures or record videos, and store the data with obscure names in the directory that is seldom visited. Then, these data are sent out to the attacker.
when Wi-Fi (fast and usually unlimited) access or other connection is available.

4.3.2 Watchdog

Watchdog is another kind of thing that a spy camera can do. Nobody wants other people to use or check his/her phone without permission. Spy camera can stealthily take pictures of the phone user and deter those who use or check other people’s phone.

4.3.3 Anti-Thief

On the other hand, spy camera could play a completely different role if it is used properly. When users lost their phones, the spy camera could be launched via remote control and capture what the thief looks like as well as the surrounding environment. Then the pictures or videos along with location information (GPS coordinates) can be sent back to the device owner such that the owner can pinpoint the thief and get the phone back.

4.4 The Basic Camera Attack Model

We want to discover possible attacks based on spy camera. The attacks should appear normal to user experiences. The main challenge is to make the attacks run stealthily and silently, so that they would not raise user’s alert. Specifically, the attacks are supposed to have translucent view, make no sound and vibration, and check phone resource utilization before launching themselves. The general architecture should include the following six parts. Figure 4.1 shows the architecture of a basic spy camera attack.

- **Step 1** To prevent user from suspecting, the malware should consider the current CPU, memory usage and battery status. Launching the attack when
CPU and memory usage are already high could make phone’s performance even worse. Users tend to doubt about the unsmooth experience and check if any app or service is running in the background. Similar concern happens to battery level, especially when the phone’s battery is low and is not being charged. Camera attack could drain the battery faster than user’s expectation and cause user suspicious about possible attacks. Hence, before launching the attack, malicious camera apps want to ensure that system resource is plenty. For Android phones, memory usage could be obtained through the `getMemoryInfo()` function of `ActivityManager`, information related to CPU utilization is available from “/proc/stat”, while current battery level and charging status can be obtained by registering a BroadcastReceiver with `ACTION_BATTERY_CHANGED`.

- **Step 2** After ensuring sufficient resources for launching attacks, a malicious camera app can continue on the remaining actions. Firstly, the app can turn off the phone’s sound and vibration, which can be achieved by setting the system sound `AudioManager.STREAM_SYSTEM` to 0 and flag to `FLAG_REMOVE_SOUND_AND_VIBRATE`. The app can log the current volume level and vibration status, and resume the parameters after the attack.

- **Step 3** The difficult task is to hide the camera preview. At the
beginning, the layout containing the $\text{SurfaceView}$ is inflated into a view via $\text{LayoutInflater.inflate()}$. Then the app can set the view parameters by changing the attributes of $\text{WindowManager.LayoutParams}$. Two important attributes must be set: $\text{TYPE_SYSTEM_OVERLAY}$ which makes the preview window always stays on top of other apps; Another one is $\text{FLAG_NOT_FOCUSABLE}$ which disables the input focus of spy camera app such that input values would be passed to the first focusable window underneath. This would turn the camera preview into a floating and not focusable layer. Then the app changes the size of preview ($\text{SurfaceView}$) to the minimum pixel (1 pixel) which human eyes cannot notice. This cannot be set directly through $\text{setPreviewSize()}$. Instead, the app needs to get the layout parameter of $\text{SurfaceView}$ by using $\text{SurfaceView.getLayoutParams()}$. Notice that the type of $\text{SurfaceView.getLayoutParams()}$ is $\text{ViewGroup.LayoutParams}$ instead of aforementioned $\text{WindowManager.LayoutParams}$. Finally, the app can add the hidden preview dynamically to the window by $\text{addView}$ function.

- **Step 4** After setting up the layout, the attack could be launched as follows: initialize the $\text{SurfaceHolder}$; choose which camera (front or back) is used and open the camera to take pictures or record videos. The photo/video data are supposed to be stored in disguises, including using confusing filenames and seldom visited directories. The app releases the camera after the above actions.

- **Step 5** After the camera attack finishes, the app sets the audio volume and vibration status back to its original values. This way, device owner would not find any abnormality.

- **Step 6** The last step of the attack is to transmit the collected data to the outside. Since cellular network usages and MMS may cause extra fees, the best choice is to wait until free Wi-Fi access is available. For example, it could use
the `javax.mail` to send the data as an email attachment. Most email systems limit the maximum size of attachments, so the length of video should have an upper bound specific to the email service.

4.5 The Remotely-controlled Real-time Monitoring Attack

The basic camera attack can be further enhanced to more aggressive attacks. For example, the attacker can remotely control the spy camera app such that the time to launch and end the attack is under control. The simplest way to implement the remote control is by socket. After the malicious app is downloaded and installed on victim’s phone, it sends a “ready” message along with the IP address and port number to the attacker’s server. Then the attacker can control the app with orders like “launch”, “stop” or specify a time schedule.

There are many Android apps that turn the phone into a security surveillance camera, i.e. Android Eye [78]. The spy camera can be easily extended to a stealthy real-time monitor based on the way of building an IP camera. NanoHttpd [79] is a light-weight HTTP server that can be installed in the phone. In our case, we can start a HTTP server at a given port which supports dynamic file serving such that the captured videos can be played online upon requests from a browser client. Figure 4.2 shows the video taken by a real-time spy camera of a mobile phone. Figure 4.2(a) is the environment that an Android phone locates. Although the phone’s screen is showing app menu, it actually captures videos through the front-face camera. Figure 4.2(b) is the view of the phone camera, which is accessed from a PC browser. The address is the IP address of the phone and the port number of the server.

In this section, we discuss the remotely-controlled real-time monitoring attack, which could pose a big threat on phone user’s privacy: daily activities and surrounding environment are all under the eye of the attacker. Camera-based attacks can be
detected when multiple apps request the camera device at the same time or if the camera is being used by another app. But this can be easily avoided by selecting the time to launch attack. The malicious camera app can periodically check the screen status and run the stealthy video recording only when the screen is off, which means that user is not using the phone and camera device is idle. The status of the phone screen can be obtained by registering two broadcast receivers `ACTION_SCREEN_ON` and `ACTION_SCREEN_OFF`.

4.6 The Video-Based Passcode Inference Attack

Since the virtual keyboard in a touch screen smartphone is much smaller than computer keyboards, the virtual keys are very close to each other. Based on the
measurement on Galaxy Nexus 4 phone, even an offset of 5 millimeters could result in touching the wrong key. Hence, when typing, users tend to keep a short distance to the screen, which allows the phone (front) camera to have a clear view of user’s eye movements. A user’s eyes move along with the keys being touched, which means that tracking the eye movement could possibly tell what the user is entering. Thus, it is of great importance to investigate whether an attacker could obtain a phone user’s passcode by tracking the eye movements.

As computer vision technique is advancing and becomes more accurate, offline processing of the video can extract the eye position in each frame and draw the path of eye movements, which means that an attacker could infer the passcode based on the video captured by spy camera app. In this section, we discuss two types of camera attacks for inferring passcode. We will also discuss the computer vision techniques for eye tracking, which can be utilized by the attacks.

### 4.6.1 The Application Oriented Attack

The first type of attack is called application-oriented attack, which targets at getting the credential of certain apps. Figure 4.3 gives some examples of app
passcodes. Most apps (like Facebook) that require authentication contain letters, which need a complete virtual keyboard as shown in Figure 4.3(a). Figure 4.3(b) and Figure 4.3(c) show two other types of popular passcodes – pattern and PIN, which we will discuss in details later. Smart App Protector is a locker app by which user is able to lock apps that need extra protection (i.e., gallery, messaging and dialing apps).

For a successful passcode inference attack, the video must be captured during user authentication. An effective way is to poll the running task list and launch the attack as soon as the target app appears on top of the list. Specifically, using the `getRunningTasks()` function of `ActivityManager`, we can get the name of most recently launched app. Meanwhile, the detection service scans the running apps and resource utilization periodically. When attack conditions are met, it opens the camera and secretly takes pictures of user’s face (especially the eyes) with front-face camera for some time long enough to cover the entire authentication process.

Besides, there are several other factors we need to consider to ensure the attack is effective and efficient: Firstly, the detection service of a spy camera app must be launched beforehand, either by tempting user to run the app or by registering a `ACTION_BOOT_COMPLETED` receiver so as to launch when booting is finished. The `RECEIVE_BOOT_COMPLETED` permission is a commonly requested permission which would not be considered dangerous. Secondly, polling task list frequently leads to an extra consumption of energy resource. To improve the efficiency of scanning, the detection service is active only when user is using the phone. As mentioned before, this can be determined by screen status. The detection service will cease when screen is off and continue as screen is lighting up again. Moreover, the scanning frequency should be set properly. In phishing attack [80], a malicious app needs to poll the running task list every 5ms to prevent the user from noticing that a new window (the fake app) has replaced the original one. In our phone camera
attack, the view is totally translucent to users so such worry is unnecessary. However, we still need to keep the frequency at around two scanning per second otherwise the attack may happen after the user starts entering the passcode (which makes the attack unsuccessful).

4.6.2 The Screen-unlocking Attack

In this subsection, we discuss another type of attack. The attack is launched when user is entering screen-unlocking passcode. We categorize this scenario as a different attack model since it is unnecessary to hide the camera preview under this circumstance.

To achieve privacy, Android system would not show the user interface until a visitor proves to be the device owner by entering the correct credential. This by accident provides a shield for spy camera attacks targeted on the screen-unlocking process. Users never know that the camera is working even though the camera preview is right beneath the unlocking interface.

We demonstrate the screen-unlocking passcode inference attack. It’s difference from the application oriented attack is the condition to launch the attack and the time to stop. Intuitively, the attack should start as soon as the screen turns on and it should end immediately as the screen is unlocked. This can be achieved with two key steps: (1) registering a BroadcastReceiver to receive *ACTION.Screen.On* when user lights up the screen and begins the unlocking process, and (2) registering another BroadcastReceiver to receive *ACTION.User.Present* when passcode is confirmed and the screen guard is gone. The second step guarantees that the camera service would stop recording and end itself immediately when the user interface is switched on. In addition, the attack should consider the situation with no screen locking passcode. To avoid being exposed, the spy camera app should check
keyguardmanager with the isKeyguardLocked() function to make sure the screen is locked before launching the attack.

To simplify the screen unlocking process, Android system provides alternative authentication methods in addition to the conventional password – pattern and PIN. Pattern is a graphical passcode composed of a subset of $3 \times 3$ grid of dots that can be connected in an ordered sequence. There are some rules for the combination of dots: (1) the number of dots chosen must be at least 4 and no more than 9; and (2) each dot can be used only once. PIN is a pure-digit passcode with length ranging from 4 to 16 and repetition is allowed. Both the two alternatives are extensively used in screen unlocking of Android phones. The relatively larger distance between adjacent keys effectively relieves user’s eye fatigue problem. However, this also brings in vulnerability to video-based passcode inference attack since larger scale of eye movement makes the attack easier.

4.6.3 Video Based Eye Tracking Techniques

In the eye tracking field, two types of imaging approaches are commonly used: visible and infrared spectrum imaging. Visible spectrum imaging passively utilizes the ambient light reflected from the eye while infrared spectrum imaging is able to eliminate uncontrolled specular reflection with active infrared illumination. Although infrared spectrum eye tracking is more accurate, most smartphones today are not equipped with infrared camera. Hence, we focus on visible spectrum eye tracking. For images captured by visible spectrum imaging, it is often the case that the best feature to track is the contour between iris and sclera known as the limbus [81].

Li et al. [81] propose Starburst eye tracking algorithm which can track the limbus of the eye. As we can see from Figure 4.4(a), in visible spectrum, they can locate where the eye is looking at in a real-time manner. However, Starburst requires calibration
by manually mapping between eye-position coordinates and scene-image coordinates. This can be performed only by the phone owner, which makes it infeasible in the spy camera attacks.

Aldrian [82] presents a method to extract fixed feature points from a given face in visible spectrum, which is based on Viola Jones adaboosted algorithm for face detection. But it is able to track pupil movement without scene image and calibration, as shown in Figure 4.4(b). We adopt this eye tracking algorithm in our research to extract eyes from videos.

4.7 Implementation and Evaluation

We have implemented the remotely-controlled real-time monitoring attack and the passcode inference attacks on real phones including the Nexus S 4G (Andriod 4.1), the Galaxy Nexus (Android 4.2) and the Nexus 4 (Android 4.3 with Security Enhanced support). For passcode inference attacks, computer vision technique for eye tracking is used to process the captured video. The evaluation of the feasibility
and effectiveness of the attacks is based on the experimental results.

4.7.1 Implementation

Both the camera based attacks are successfully implemented on Android phones equipped with a front-face camera. The spy camera apps are completely “translucent” to phone users and work without causing any abnormal experiences. When Wi-Fi access is enabled, the captured data is transmitted to the attacker via a local HTTP server or email. To test the stealthiness of the attacks, we install two popular antivirus apps: AVG antivirus and Norton Mobile Security. Neither of the two antivirus apps has reported warning during the entire video capturing and transmission process. This demonstrates their resistance to mobile anti-virus tools.

4.7.2 Feature Analysis of the Passcode Inference Attack

An important feature that enhances the effectiveness of passcode inference attack is that it could be launched in repeated manner, which allows certain passcode to be “attacked” many times. In this way, attacker could get a set of possible passcodes and keep launching attacks until the correct one is found.

The passcode inference attack depends on the victim’s eye movement instead of analyzing videos containing the screen [76] or its reflection [77], which makes it harder to achieve high and stable one-time success rate. In addition, there are complex factors that may influence its performance, such as the distance between face and phone, lighting condition, velocity of eye movement, pause time on each key and head/device shaking when typing. Among these experimental conditions, only the lighting condition can be kept constant during our experiments.

To test the effectiveness of the passcode inference attacks with different types of passcodes, we use the conventional password, pattern and PIN in our experiments.
By comparing the rules of pattern and PIN, we find that pattern combination is actually a subset of PIN. In addition, the outlines of the two passcodes are similar (both are squares). Hence, we present their results and discuss their performance together. Another consideration for experiments is the length of pattern and PIN. In fact, people rarely use long PIN and complex patterns since it’s hard to memorize and impractical for frequent authentications such as screen unlocking. This can be best illustrated by Apple iOS’s 4-digit PIN for screen unlocking. Hence, in our experiments we choose 4-digit pattern/PIN for testing.

4.7.3 Performance Evaluation

We process videos containing user eye movement with the aforementioned computer vision technique [82]. Due to the tight configuration of virtual keyboard and limitation of visible spectrum imaging, the performance of inferring conventional password is poor and unstable. However, the possibility of compromising pattern and PIN is shown to be much higher. In our evaluation, 18 groups of 4-digit passcodes were tested, and the results are listed in Table 4.1.

In Table I, each group consists of three components: real passcode (Real Psscd), eye movement (Eye Movement) and possible passcodes (Possible Psscds). Since the shape of the 9-dot pattern keypad and 10-digit PIN keypad is similar to a square, the results of eye movement are drawn in a square. Therefore, position projection can be used to infer input keystrokes.

We find that in some cases, the correct passcode can be inferred accurately. While in other cases, it is in a small group of possible passcodes. For example, from the first row in Table I, the passcode “1459” can be directly inferred while “1687” and “1450” both have three candidates. The attacker could further narrow down the possible passcode set by launching more attacks and finding out the intersection.
Table 4.1: Passcode Inference Results for 4-digit Passcodes

<table>
<thead>
<tr>
<th>Real Psscd</th>
<th>Eye Movement</th>
<th>Possible Psscds</th>
<th>Real Psscd</th>
<th>Eye Movement</th>
<th>Possible Psscds</th>
<th>Real Psscd</th>
<th>Eye Movement</th>
<th>Possible Psscds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1459</td>
<td><img src="1459" alt="Graph" /></td>
<td>1459</td>
<td>1687</td>
<td><img src="1687" alt="Graph" /></td>
<td>1450</td>
<td>1450</td>
<td><img src="1450" alt="Graph" /></td>
<td>1487</td>
</tr>
<tr>
<td>1486</td>
<td><img src="1486" alt="Graph" /></td>
<td>1486, 14786, 1786</td>
<td>1479</td>
<td><img src="1479" alt="Graph" /></td>
<td>1856</td>
<td>1856</td>
<td><img src="1856" alt="Graph" /></td>
<td>1450, 1458, 2569</td>
</tr>
<tr>
<td>1359</td>
<td><img src="1359" alt="Graph" /></td>
<td>1359, 13659, 13589, 136589, 12359, 123659, 123589, 1236589</td>
<td>2548</td>
<td><img src="2548" alt="Graph" /></td>
<td>2548, 3659, 1793</td>
<td>1793</td>
<td><img src="1793" alt="Graph" /></td>
<td>1450, 1452, 2563, 4785, 5896</td>
</tr>
<tr>
<td>1953</td>
<td><img src="1953" alt="Graph" /></td>
<td>1953, 15953</td>
<td>2856</td>
<td><img src="2856" alt="Graph" /></td>
<td>2856, 1745</td>
<td>1759</td>
<td><img src="1759" alt="Graph" /></td>
<td>14759</td>
</tr>
<tr>
<td>4734</td>
<td><img src="4734" alt="Graph" /></td>
<td>4734, 47534, 47354, 475354</td>
<td>1490</td>
<td><img src="1490" alt="Graph" /></td>
<td>1490, 1790</td>
<td>1595</td>
<td><img src="1595" alt="Graph" /></td>
<td>1595</td>
</tr>
<tr>
<td>2450</td>
<td><img src="2450" alt="Graph" /></td>
<td>2450, 2480, 2458, 3569, 3469</td>
<td>1741</td>
<td><img src="1741" alt="Graph" /></td>
<td>1741, 14741, 2852, 25852, 5085</td>
<td>1865</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: The table includes visual graphs for each passcode, indicating the possible movements and choices made during the process.
Furthermore, when the owner is not using the phone, the attacker may try different possible passcodes and see which one works.

4.8 Countermeasures

In this section, we discuss possible countermeasures that can protect Android phones against these spy camera attacks.

In Android system, no API or log file is available for user to check the usage of camera device. Hence, detection of the camera based attacks requires modification to the system. We make changes to the `CheckPermission()` function of `ActivityManagerService` and write a lightweight defense app such that whenever camera is being called by apps with CAMERA permission, the defense app will be informed along with the caller’s Application Package Name. The Application Package Name is an unique identifier in Android and third party apps cannot reuse the name of built-in apps like Camera (com.android.gallery3d). By looking at the app name, we are able to identify the built-in camera app (that is known to be safe) and no alert will be generated.

Then, we design further checking mechanism for 3rd-party camera based apps. Analyzing activity pattern is an effective approach in malware detection. For each type of spy camera attack, we are able to extract specific feature from its activity pattern. The activity pattern of spy camera apps are totally different from legitimate camera apps. To avoid collision in the use of camera, remotely-controlled real-time monitoring attack runs when the screen is off. The application-oriented passcode inference attack launches immediately after other app runs. The screen-unlocking attack runs when screen turns on but remains locked. The defense app is able to decide the dynamic launch pattern of camera related apps by polling the task list. If a camera app calls camera in one of the above situations, the defense app would give
warnings to the phone user.

There are three parts of warnings in our defense scheme. Firstly, an alert dialog including the name of suspicious app is displayed. In case that the warning message could not be seen immediately by the user (e.g., the user is not using the phone). The defense app will also make sound and vibration to warn the user of spy camera attacks. Besides, the detailed activity pattern of suspected apps are logged so that the user can check later. As shown in Figure 4.5, the spy camera app named \texttt{com.example.as.bakvideo_lock} calls camera as soon as Facebook is launched. This process is detected by the defense app, and warning message with its name is displayed before the user enters credential.

### 4.9 Chapter Summary

In this chapter, we study camera-related vulnerabilities in Android phones for mobile multimedia applications. We discuss the roles that a spy camera can play to attack or benefit phone users. We discover two advanced spy camera attacks, including the remotely-controlled real-time monitoring attack and the video-based
passcode inference attack. Finally, we propose an effective defense scheme to secure smartphone from all these spy camera attacks.

Our major contributions can be summarized as follows:

- We comprehensively study the camera-based attacks on smartphones. We first present the basic spy camera attack model with a step-by-step analysis.

- Then, we further develop two advanced camera attacks: the remotely-controlled real-time monitoring attack and the video-based passcode inference attack. The remotely-controlled real-time monitoring attack allows the remote attacker to stealthily spy on the phone user in real-time. The video-based passcode inference attack intends to obtain the password from the captured video that contains the user eye movements while entering password.

- We implement the attacks and the countermeasure we designed on an Android smartphone. The experiments show that the video-based passcode inference attack is feasible when attacking simple password. In addition, the proposed defense scheme is demonstrated to be lightweight and effective.
CHAPTER 5

TWO FACTOR AUTHENTICATION SCHEME
FOR INTERNET-OF-THINGS

While the rapid development of IoT devices is changing our daily lives, some particular issues hinder the massive deployment of IoT devices. For example, current network ID management system cannot handle so many new terminals; there is no agreed security standards for IoT manufacturers to follow when designing their products. The whole IoT industry is expecting the breakthrough in network infrastructure and the development of novel security mechanisms that can enable the flexible, secure and reliable access and management of IoT devices. Bitcoin, first released in 2009, breeds the decentralized Blockchain technology. The decentralization, anonymity and proof of security characteristics of Blockchain, shed new light on the evolution of IoT system. We believe that the application of Blockchain into the IoT system can clear the obstacles facing the development of IoT architecture and security. To this end, we propose an out-of-band two-factor authentication scheme for IoT devices based on Blockchain infrastructure. We implement the IoT and Blockchain integrated system with Eris Blockchain and equivalent computing devices to emulate IoT devices. The overheads to run Blockchain and smart contract services on the emulator devices are measured. The BeagleBone Black and Raspberry Pi 3 nodes have an average memory usage of 29.5M, and the CPU usage of 29.55% and 13.35%, respectively.
5.1 Introduction

Internet-of-Things (IoT) is a network of interconnected objects embedded with electronics, software and sensors. IoT devices are gaining growing popularity on consumer electronics market in the last few years. Increasingly more IT companies have added their IoT products line. For example, Apple released its first smart watch [83] in April, 2015; Nest Labs (acquired by Google in 2014) released its 3rd generation learning thermostat [84] in September 2015; in October 2015, Philips released a new smart light network bridge [85] that allows users to control the Philips Hue lights using mobile apps. According to [86], the total amount of the “connected things” will reach 20.8 billion by 2020. Needless to say, it’s a huge market and a field of promising technology innovation.

However, IoT devices could suffer a variety of malicious attacks, including both the traditional Internet attacks and the specifically designed attacks targeted at IoT devices. The traditional Internet threats they are facing include message replay, impersonation, and man-in-the-middle attacks that can violate the authenticity principle of security. They may also suffer the denial of service (DoS) attacks can prevent legitimate users from being successfully authenticated. IoT devices are more vulnerable to these attacks compared with computers, due to their limited computational and memory resources. Meanwhile, IoT devices are also susceptible to IoT-specific attacks such as in-home smart appliances [11,87].

There is no common security standards for IoT devices. The commercial IoT devices on the market are protected by the security schemes designed by each individual manufacturers. This poses huge potential safety hazard since instead of boosting the sales of their products, developing and implementing more robust security mechanisms could only lead to the increase of cost and complexity. The numerous IoT security flaws reported [88,89] all warn us that the existing protection
provided by the manufacturer is far from sufficient. Traditional authentication by password requires human interaction. To enhance the security, two-factor authentication schemes taking advantage of SMS and phone calls require additional infrastructure and human involvement.

The Bitcoin system [90] is peer-to-peer without a central repository or administrator. All transactions take place between users directly. The transaction data are cryptographically secured and maintained by “miners” all over the world. Miners are computer nodes that contribute their computing resources to help maintain and verify bitcoin transactions. As incentives, miners who generate the next transaction block will be rewarded with some bitcoins which they can redeem for money through online bitcoin exchanges. The Blockchain is a distributed database that maintains a continuously growing list of ordered records called blocks. As the core component of the bitcoin system, Blockchain serves as the public ledger for all transactions. By design, Blockchain allows add-only transaction data and prevents the tampering of historical data.

Home IoT devices may need to share data or deliver commands to each other. To secure the access of home IoT devices, we propose a novel two-factor authentication scheme based on Blockchain technology. The secondary authentication is conducted over an out-of-band channel (light, acoustic, etc.), by another friend IoT device within the home other than the one being accessed (subject device). The friend device will verify whether the access requestor is inside the home, and securely send the result to the subject device via Blockchain.

5.2 Adversary Model

In our work, we mainly consider the household scenarios, in which a group of different IoT devices collaborate (e.g., share their sensed data) to better serve the
house owner. However, to preserve the privacy of sensitive data, the requester device must first authenticate itself to the provider device and gain the authorization to access the data. To better describe the threat model, we use the automatic light control based on Nest learning thermostat and Philips Hue Smart Bulb as an example. Specifically, we use Nest thermostat as the ambient light condition and temperature data acquisition device and the Philips Hue smart bulb as the light controller. The thermostat sends the measured data to the Nest server, and the smart bulb access the data from Nest server to adapt its color and brightness to the change of ambient light condition and temperature. For example, the Hue bulbs automatically increase brightness when the Nest device senses dim ambient light and change to warm color when the room is cold.

In order to achieve the auto-adaptation of brightness and color, the Philips Hue bulb first needs to authenticate itself to the Nest server and obtain the sensing results. Figure 5.1 describes the light auto-adaptation use case. The nest server can be reached
and verified by checking the certificate issued by certificate authenticators such as Symantec and Trustwave through the SSL/TLS communication. The Nest API uses the OAuth 2.0 protocol for authentication and authorization. Before an IoT product of other brands can access the private data via the Nest API, it must obtain an access token granted for access to that API. For the smart bulb, in order to be authorized to access the Nest server, it has to register to the Nest server and get the authorization code from the authorization server. The user grants an authorization by logging into the account and accepting the privacy and permission agreements using a browser or the smart bulb mobile application, as shown in Figure 5.2. After the user pressed the ACCEPT button, an access PIN code will be issued for the smart bulb, as shown in Figure 5.3, which will be exchanged to the access token to access the Nest server.

The process that the user obtains and enters the access code to the smart bulb
mobile application leads to a relatively vulnerable channel between the bulb and the Nest server, considering the following potential compromises:

1. The attacker launches phishing attack to steal PIN code.
2. The attacker physically approaches the user to oversee the PIN code during the initial setup process.
3. The user may store the PIN code in an insecure storage, which may be obtained by an attacker.

Given these potential attacks, serious data leakage could happen. In the use case shown in Figure 5.1, the attacker can impersonate to be the real smart bulb and gain access to the data using the stolen access code. Our proposed scheme focuses on the defense to an external adversary who has compromised the access token in home IoT scenarios.

5.3 The Out-of-band Two-factor Authentication Scheme

In this section, we present our out-of-band two-factor authentication scheme to enhance the authentication and authorization process. Apart from the existing
Figure 5.4: The Out-of-band Secondary Authentication

authentication on commercial IoT devices, our scheme takes advantage of the out-of-band channel to conduct a secondary authentication. The secondary authentication factor is able to distinguish a home IoT device from the malicious device (e.g. software-defined radio controlled by the attacker) outside the house, even if the malicious device impersonates the legitimate IoT device using the correct access token.

Following the aforementioned light auto-adaptation use case, we define the four entities considered in our scheme as follows (depicted in Figure 5.4):

1. **Authentication Subject (AS)** is the device that needs to be authenticated in order to access services or data, which is the Philips Hue Bulb in the aforementioned case.

2. **Related Device (RD)** is the verifier device for the secondary authentication. We assume that RD and AS has been mutually authenticated with each other. In the aforementioned use case, the RD can be any other smart home devices (e.g. WeMo, Belkin, etc.).

3. **Blockchain** stores the relationship information for each AS, namely the RDs
that are able to serve as the authenticator in an out-of-band channel. It also provides storage for the verification results.

4. **Authentication Executor (AE)** is the entity that performs and coordinates the two-factor authentication. The Nest server is the AE in the light auto-adaptation use case.

The core idea of the out-of-band secondary authentication is to verify whether an access requestor AS locates within the home (i.e., physically near its RDs) or not. Specifically, AE sends a sequence of action code (bit stream) to AS, which will then flash according to the sequence of action code (e.g., 1 for ON and 0 for OFF), in a high frequency that human eyes cannot notice. Meanwhile, AE chooses one or multiple RD(s) of AS to confirm that the action has been correctly executed by AS. Each RD has an established relationship (e.g., in proximity) with AS, and can accurately measure the bit stream embedded in the changing of lighting conditions. In contrast, the outside adversary have no control over the indoor lighting conditions, hence the RD(s) will not get the right action code and the adversary will fail the secondary authentication. The verification result will be recorded on the specific address on Blockchain. Finally, AE will read RD’s verification result to decide whether to grant the access request of AS. Similarly, the secondary authentication can also be performed on other out-of-band channels (e.g., using acoustic signals).

Figure 5.4 presents the out-of-band (light) secondary authentication. The detailed procedure is listed as below:

1. AS (Philips Hue bulb) requests for the room temperature from AE (Nest server) so that it can changes color based on the current temperature.

2. AE retrieves the relationship information of AS from Blockchain.

3. AE selects the RD (WeMo Link) that is in close proximity with AS, then it sends the action sequence to AS.
4. AS receives and executes the action sequence by switching the bulb on/off accordingly (as the light signal).

5. The proximity RD decodes the code embedded in the light signals.

6. Proximity RD sends the verification result to Blockchain by invoking function of the Smart Contract on Blockchain.

7. AE checks Blockchain for the verification result through smart contract.

### 5.4 Implementation and Evaluation

#### 5.4.1 Evaluation Environment Setup

Since the aforementioned commercial IoT devices use closed-source hardware and software systems, we are not able to modify and add our authentication scheme onto those devices. Instead, we set up our experimental environment on other developer-friendly devices, which have the similar processing power to WeMo devices and the Philip Hue bulb. Table 5.1 shows the emulator devices we used (Column 2) together with their specifications (Column 3), in contrast to the real-world IoT devices (Column 1).

Figure 5.5 illustrates the experimental setup. The emulator devices we used are Beagle Bone Black and Raspberry Pi 3 boards, both are widely-used

<table>
<thead>
<tr>
<th>IoT Device</th>
<th>Emulator Device</th>
<th>Specification</th>
</tr>
</thead>
</table>
| Philips Hue bulb   | BeagleBone Black| • 1GHz ARM Cortex A8 processor  
                      |                  | • 512 MB DDR3 RAM and 4GB 8-bit embedded MultiMediaCard (eMMC)  
                      |                  | • On-board flash storage  
                      |                  | • USB Wi-Fi Dongle |
| Nest data server   | Raspberry Pi 3  | • 1.2 GHz ARM Cortex-A53  
                      |                  | • 1G RAM  
                      |                  | • On-board Wi-Fi and Bluetooth |
| WeMo Link          | Raspberry Pi 3  | Same as above.                  |
| WeMo Switch        | Raspberry Pi 3  | Same as above.                  |
prototyping platforms in Internet-of-things and embedded systems. They have Wi-Fi communication interface built on board and can connect to the Internet and the Blockchain. The mutual authentications between AS and its related devices ($RD_1$, $RD_2$) have already been completed, and their relationship information have been posted to the Blockchain.

5.4.2 Relationship Establishment on Blockchain

There are two parts for the relationship establishment on Blockchain: the device profile and the pairing of related devices. As in Eris Blockchain implementation, the two parts correspond to two separate smart contracts: one is the device contract and the other is the relationship contract. The device contract stores the device profiles and the relationship contract stores the related device pairing information. Figure 5.6 shows a sample of the smart contract execution logs.

As for the device contract, the device profiles are stored in a mapping data structure
The profile data includes the following information:

- Device name
- Device address
- Device registration date
- Wireless I/F (Wi-Fi, Bluetooth, etc.)
- Resources (Light sensor, microphone, speaker, etc.)

Likewise, the relationship contract contains the mapping of the AS address to the relationship data, including the following information:

- Related device address
- Proximity (optional)
- Relationship establishment date
- Relationship expiration date
5.4.3 Blockchain Performance

To evaluate the performance of the Blockchain based authentication scheme, we measured the memory and CPU usage of each node in the system. These results are obtained by using the “top” command of Linux to get a runtime screenshot of the Blockchain process every 10 seconds and calculating the average value of 100 consecutive screenshots. Figure 5.7 shows the results of the memory usage of the Blockchain process. The memory consumptions on the Beaglebone Black board and Raspberry Pi 3 are similar. The average memory usage for running the Blockchain is around 29.5 MB, which is trivial compared to their total memory resources (512M and 1G). Table 5.2 presents the CPU usage of the Blockchain process. The percentage of CPU overhead on BeagleBone Black and Raspberry Pi 3 nodes in average are 29.55% and 13.35% respectively, which are considered acceptable since it happens only in the authentication process.
5.5 Literature Review

There are related works on multi-factor authentication, multi-channel authentication and Blockchain based authentication. However, none of them was designed for large scale IoT devices.

Morfrio et al. [91] proposed a novel location-based second-factor authentication solution for modern smartphones, which employs TEE (Truest Execution Environment) to enforce security channel for location exchange. The authors discussed user identification enrollment based on signed-IMSI and Baseband OS. As for security, the location based second factor authentication can achieve the detection of fraud attacks, in which the attacker holds the card and the PIN but does not have access to the victim’s phone.

Aboshosha et al. [92] proposed an efficient one time password (OTP) based authentication protocol over a multi-channel architecture. The purpose of the protocol is to integrate a web based application with mobile-based technology to communicate with the remote user through a multi-channel authentication scheme. By integrating a Web-based application with the mobile-based technology, the proposed protocol can overcome many challenging attacks such as replay attack, DoS attack, man-in-the-middle (MITM) attack, real-time phishing (RTP) and other malicious attacks. To prevent the OTP from eavesdropping attack, the proposed authentication method adopts RC4-EA encryption method and the QR-code technique.

Another paper [93] that studied the multi-channel authentication uses mobile phones as the second channel. The authentication is performed in the following manner: (1) A user reads the session ID of a communication channel between a service provider and a PC using a barcode reader on the mobile phone terminal and (2) sends the session ID through mutual authenticated secure channel over a mobile
phone network to the authentication server and (3) the authentication server matches the session ID and binds the user with the corresponding communication channel to provide service to the PC.

Garman et al. [94] proposed a novel anonymous credential scheme without the need of a trusted credential issuer. Specifically, a public append-only ledger (Blockchain) is employed to conduct the authentication. The authors provided a proof of security for an anonymous credential system that allows users to make flexible identity assertions with strong privacy guarantees without relying on trusted parties. They exploit namecoin, a system built on top of bitcoin Blockchain to provide name-value mappings, to store public key and the corresponding credential.

Only one recent work has studied the secondary authentication in IoT context. Griffin [95] discussed the idea of utilizing biometrics for authentication such as speaker recognition, hand gesture, and gait biometrics. A knowledge-based authentication using both password strings and data extracted from biometric sensors is proposed. For example, suppose voice is chosen as the biometric, then the first authentication factor “something-you-know” comes from the words spoken by a user, and the second authentication factor “something-you-are” is the biometric matching data (i.e., speech recognition). While biometrics are characteristics for human identification, our scheme focuses on the authentication of IoT devices instead of the user/owner.

The combination of Internet-of-things and Blockchain has been studied in another recent work [96], which examines whether the Blockchain technology makes a good fit for the IoT sector. The authors found that Blockchain can facilitate the sharing of services and resources between IoT devices and allow users to automate several existing time-consuming workflows in a cryptographically verifiable manner. However, this work only highlighted the Blockchain’s benefits in automating the interactions between transacting parties (e.g., transactions), and in developing new business
models. Instead, our work takes advantage of Blockchain to assist the authentication of IoT devices.

5.6 Chapter Summary

Securely and reliably authenticating the large scale Internet-of-things devices is not trivial with current network framework and existing security mechanisms. We proposed an out-of-band two factor authentication scheme that exploits device relationship, supported by Blockchain technology. The secondary authentication of our scheme can prevent the access of external malicious devices, even if the first factor fails (e.g., the access token is stolen by the adversary). The out-of-band channel can be either light or audio. The device relationship information is stored on Blockchain, which is resistant to collusion and single point failure of the centralized server. We implement the proposed authentication scheme with emulator devices that have similar computational resources as real IoT devices and commercial Blockchain platform. The performance of the emulator devices running Blockchain tasks is evaluated. The experimental results indicate that the CPU and memory overheads are well acceptable, considering they occur only during the authentication phase.

We summarize our contributions below:

- To the best of our knowledge, this is the first work to use Blockchain to assist the authentication of IoT devices.

- We design a secondary authentication over an out-of-band channel (light, acoustic, etc.), which can detect outside malicious devices even if they have stolen the passcode or the access token.

- Our scheme uses only IoT devices within the home for the secondary authentication. No private information is leaked to external devices.
• We set up the experimental environment with commercial Blockchain and emulator devices. The performance of devices running our proposed scheme is evaluated.
CHAPTER 6

SMARTPHONE-BASED ACCESS CONTROL SCHEME FOR IMPLANTABLE MEDICAL DEVICES

With the rapid development of health equipment, increasingly more patients have installed the implantable medical devices (IMD) in their bodies for diagnostic, monitoring, and therapeutic purposes. IMDs are extremely limited in computation power and battery capacity. Meanwhile, IMDs have to communicate with an external programmer device (i.e., IMD programmer) through the wireless channel, which put them under the risk of unauthorized access and malicious wireless attacks. In this chapter, we propose a proxy-based fine-grained access control scheme for IMDs, which can prolong the IMD’s lifetime by delegating the access control computations to the proxy device (e.g., smartphone). In our scheme, the proxy communicates with the IMD programmer through an audio cable, which is resistant to a number of wireless attacks. Additionally, we use the ciphertext-policy attribute-based encryption (CP-ABE) to enforce fine-grained access control. The proposed scheme is implemented on real emulator devices and evaluated through experimental tests. The experiments show that the proposed scheme is lightweight and effective.

6.1 Introduction

IMDs are the particular type of medical devices that are implanted in the patient’s body, to diagnose, monitor, or treat a variety of conditions, diseases and injuries. For
example, insulin pump can monitor and deliver insulin to treat diabetes, pacemaker regulate the beating of the heart using electrical impulses, neurostimulator sends electrical signals to the spine to treat chronic pains. According to a recent report published by Allied Market Research [97], the global IMD market is projected to reach $116.3 billion by 2022. However, IMDs are threatened by both external cyber attacks and internal flaws in software or firmware design. These security vulnerabilities allow an adversary to steal sensitive medical data, reset the configuration parameters, and issue unauthorized commands to an IMD, which could cause fatal consequences.

IMDs are equipped with a radio transceiver to communicate with the external IMD programmer. The IMD programmer is the specific device used to collect the medical data from IMDs and issue operation/configuration commands to deliver drug, change dosage, etc. With the wireless interface enabled, IMDs can be accessed by an authorized operator in physical proximity via the IMD programmer (e.g., an eligible medical staff or the patient himself/herself). However, the wireless communication and networking capabilities of IMDs turn out to be the major sources of security vulnerabilities. Due to the broadcast nature of wireless channels, all messages exchanged between the IMD and the programmer can be captured by eavesdroppers. This would not only expose the patient privacy like he/she is carrying an IMD to treat a certain disease, but also lead to other classic wireless attacks such as the forging, tampering, and replying of the messages. Existing research works have presented the breaches in a number of commercial IMDs [12–16], including the implantable cardioverter defibrillator (ICD), insulin pump and pacemaker. It has been demonstrated how an adversary can reverse-engineer the communication protocol and take full control of the IMD using a software radio.

Intuitively, to secure the communications between the IMD and the IMD programmer, a pair of symmetric keys must be shared between the two parties to encrypt their wireless communications. Unlike traditional electronic devices, the
power supply of IMDs is highly constrained. The wireless charging technologies for
IMDs still need lots of practical testing and clinical trials to ensure that no negative
effect will be caused to human organs and tissues. Additionally, the replacement of
an IMD or its battery requires invasive surgery. Hence, commercial IMD products
are designed to last for 5 to 10 years. The energy consumption of IMDs should
be minimized, by avoiding complicated cryptographic computations and long-range
wireless communications. Currently, only symmetric cryptography is considered for
the data encryption in IMDs.

In this chapter, we present a novel proxy-based access control scheme for IMDs,
in which the communications between the IMD programmer and the proxy are
conducted through an audio cable rather than the conventional wireless channels.
The proposed scheme employs the attribute-based access control model, which grants
access based on the attributes (i.e., qualifications) that the access requestor owns.
Meanwhile, the access requestor is authenticated in our scheme, mainly to provide
accountability in case of a medical dispute.

6.2 Motivation

The security researchers have been seeking generalized and effective access control
schemes for IMDs. A variety of access control schemes have been proposed with
different access control models and architectures. In addition, various assumptions
have been made on the environmental settings and human factors. In this section, we
conduct a thorough analysis on all these aspects, and present how we are motivated
to design our novel IMD access control scheme.
6.2.1 Special Use Cases

IMD access control is not a difficult problem under regular situations such as when the patient uses his/her own IMD programmer to access the IMD or when an acquainted doctor wants to access the IMD. However, it becomes much more complicated in some special but practical situations.

6.2.1.1 Medical Emergency

In medical emergency conditions such as when the patient falls sick while travelling out of town and needs immediate treatment, the patient has to verify the authenticity and qualification of the stranger who attempts to access the IMD (e.g., emergency medical technician). In a worse case, the patient may have been unconscious and is not able to manually verify the programmer operator. Hence, to deal with the emergency cases, an effective IMD access control scheme requires:

1. All eligible medical personnel should have access to the IMD, regardless of whether they have been granted access before or if the patient is acquainted with them.
2. The access decision can be made autonomously by the IMD, without the patient’s involvement.
3. If the access is permitted, the IMD must first be paired up with the programmer so that a pair of symmetric keys are shared between them, to encrypt their communications.

6.2.1.2 Internet Connection

Online authentication has been widely used in IT applications. Intuitively, offloading authentication to a dedicated server of a governmental health agency or hospital can greatly reduce the complexity of the access control computations running
This requires either the IMD (may be assisted by an external proxy) or the programmer to be able to connect to the Internet. However, Internet connection may not be always available, especially when the patient is located in a depopulated area with poor infrastructure. Hence, a robust IMD access control scheme should not rely on online authentication.

6.2.2 Adversary Model

The common assumptions agreed by existing works regarding the capabilities of the attackers in IMD context include:

- The adversary may be equipped with powerful software radio transmitter, hence is able to interact with the IMD in a long distance.

- The adversary may obtain legitimate IMD programmer to access the IMD. Since IMD programmers are specialized devices running closed-source software programs, they are considered secure and cannot be hacked by the attacker.

- The adversary cannot approach the patient within a security range (typically 10 cm), nor can the adversary make physical contact with the patient or the patient’s personal belongings, deterred from leaving criminal evidence such as fingerprint or video taken by surveillance cameras.

Generally, two types of adversaries may exist depending on the attack tactics: passive adversary and active adversary. A passive adversary will only eavesdrop on the wireless channel and listen to the packets exchanged between the IMD and the IMD programmer. Given an unencrypted radio channel, a passive attack can break the confidentiality of the data being transmitted. Almost all existing access control schemes require encryptions over the wireless channel, hence are resistant to passive attackers. An active adversary, however, can replay or tamper the packets. If the communication protocol between the IMD and the programmer is reverse-engineered,
the active attacker is able to send unauthorized commands to the IMD (e.g. changing the configurations and parameters). Based on the purposes of the attacks launched by an active adversary, we can classify active attacks into three categories:

- **Unauthorized access.** The goal of this type of attacks is to bypass the access control scheme and gain access to the IMD without authorization.

- **Resource depletion.** This type of attacks repeatedly requests for access to the IMD, causing the IMD to be continuously running the access control computations, while its actual intention is to drain the battery power and reduce the lifetime of the IMD.

- **Denial-of-service (DoS).** DoS attacks aim to disrupt the authorized access to the IMD, by interfering the communications between the IMD and the programmer. As the result, the IMD is unable to serve the incoming requests.

The resource depletion attacks and DoS attacks are not the focus of our work. Many previous works have proposed effective schemes to solve the resource depletion attacks on IMDs. Liu *et al.* [100] suggested to add an extra wake-up circuit before the main circuit of the IMD, which employs the passive RFID technology so as to harvest energy from the incoming signal to perform the verification of the wake-up code. The main circuit is waked up only if the wake-up code is correct. Gollakota *et al.* [101] presented *Shield* to protect the confidentiality of the IMD, which utilizes a novel full-duplex radio design with a jamming antenna and a receive antenna, allowing it to simultaneously receive the IMD’s signal and jam the IMD’s messages. Consequently, the programmer cannot receive IMD’s packets or directly interact with it. Hei *et al.* [102] proposed to train the normal IMD access patterns and detect unusual access requests using the support vector machine (SVM). The DoS attacks, however, have been evaded in existing works. Since the IMD is installed inside human body and has to interact with the programmer over a wireless channel, an attacker can just
block/interfere its communications by jamming the wireless channel. Although the DoS attacks can be easily detected, there is no effective and low-cost solution to prevent it.

Instead, our work focuses on enhancing the IMD access control in terms of lower complexity and better granularity.

6.2.3 Access Control Architecture

6.2.3.1 Two-party Access Control

The basic IMD access control architecture is composed of two parties. As illustrated in Figure 6.1(a), the access object is the IMD and the subject is the programmer and its operator.
6.2.3.2 Proxy-based Access Control

To reduce the energy consumption of the IMD, the proxy-based access control architecture has been proposed [102–105], which take advantage of a proxy device to delegate the heavy computations for the IMD. As shown in Figure 6.1(b), the proxy can be a smartphone or other wearable device (e.g., smart watch, smart bracelet) with more sufficient computational resources and battery capacity. The communications between the IMD and the proxy are protected by the lightweight symmetric encryption, which can be considered safe given that a pair of symmetric keys have been distributed and shared securely during the initial setup. This is reasonable since the initial setup is conducted either by the doctor when the IMD is implanted or by the patient when the proxy is used for the first time. It is very difficult for a malicious eavesdropper to overhear the key being transmitted to IMD at these specific moments. After pair-up, the proxy device will perform the access control on behalf of the IMD.

The proxy-based access control depends on the presence of the proxy device. In the particular case that the proxy is not detected in vicinity, the commonly used solution is that the IMD will enter the open-access mode, in which it only verifies the physical proximity of the programmer and permits the incoming access requests from any programmer nearby. This allows eligible physicians to be still able to access the IMD when the proxy is damaged or lost. Later, patient can pair up the IMD with a new proxy using its unique master key. A copy of this master key is provided to the patient by the IMD manufacturer along with the product manual. Meanwhile, the IMD manufacturer or the hospital should keep a backup copy of the master key for the patient to retrieve from.

However, we are aware that the usage of a proxy device may increase the attack surface of the IMD access control. Two types of attacks targeted at the proxy may
exist:

- **Jamming attacks.** The adversary may attempt to bypass the access control performed by the proxy, by selectively jamming the messages of the proxy to spoof the IMD about the absence of the proxy.

- **Malware-based attacks.** If the proxy is a general-purpose device like smartphone, the adversary can attack through a pre-installed malware. However, since Android OS and iOS both use sandboxing to isolate applications from each other, the malware cannot compromise the client application running the access control program. Instead, it can only eavesdrop or disrupt the communications between the proxy and the IMD/programmer.

Our scheme employs the proxy-based access control architecture. Note that we do not specifically address the jamming-based spoofing attack - our access control scheme can nicely integrate with existing solution [104]. Additionally, we do not propose new access control method for the particular case in which the proxy is indeed absent. The existing coarse-grained proximity-based access control schemes [106–111] can be adopted to protect the IMD under such circumstances.

### 6.2.4 Access Control Model

There are three types of traditional access control models that may be applied in IMD context.

- **Identity-based access control (IBAC):** a user is permitted or denied access based on whether the user appears in the access control list that contains all of the authorized users.

- **Role-based access control (RBAC):** the access permission is granted to a group of people who have a common role.
• **Attribute-based access control (ABAC):** a set of attributes are created and assigned to subjects to enforce the access control. The access rule is defined as a mixing of attributes, and the decision is made by matching the attributes required.

In fact, most existing access control schemes are not designed under traditional models, but instead are simplified to accommodate the resource-limited IMD:

• **Pre-shared secret based access control:** Some works assume that the IMD and the programmer have pre-shared secret like a master key [12] or rolling code [13]. However, considering that the IMD may be accessed by any doctor in emergency situations, the pre-distribution of the secret of each IMD to all possible doctors is not practical.

• **Proximity-based access control:** Some existing schemes manage access control solely based on physical proximity [106–113]. Although proximity is sufficient under the adversary model that the attacker will not approach the patient or make physical contact, it can only provide coarse-grained access control in which the identity of the requesting programmer operator is not authenticated.

To enable the authentication of the programmer operator, public-key cryptography must be used, which is feasible in the proxy-based access control architecture. However, considering that online authentication is unavailable and that it is not practical to store the information of the tons of eligible medical personnel into a local ACL on the proxy, IBAC model is not a viable option in IMD context. Instead, ABAC allows the multi-dimensional rules/policies (not just based on the identity or a simple role) to be enforced for fine-grained access control such as the specialty and affiliation of the programmer operator, the certificate of the eligibility to operate a certain model of IMD, etc.

Therefore, our scheme employs the ABAC model to verify the qualifications of the access requestor. Meanwhile, to provide accountability in medical disputes, the proxy
also authenticates the programmer operator and (if authorized) will record the access details (i.e., start time, end time) into a log as the evidence.

6.2.5 Communications through Audio Cable

In the proxy-based access control architecture, the proxy needs to first build a secured connection with the IMD programmer. However, for general-purpose proxy device like smartphone, setting up a local connection with an external device (e.g. IMD programmer) via Bluetooth, NFC, or USB requires either the phone to be in the unlocked mode or the manual approval on the smartphone (also when the phone is unlocked). This means such types of connections will be unavailable when the smartphone is locked and the patient has gone unconscious, as the programmer operator does not know the password to unlock the patient’s phone. According to the recent data from Duo Labs, 34% Android smartphones are not secured with a lock-screen passcode [114]. In another word, about 2/3 of phone users have enabled the screen-locking function. Therefore, we must choose a connection channel that is secure and available regardless of the patient’s involvement.

We found that most modern smartphones have a headphone jack/port and most commercial IMD programmers have a USB port. A smartphone serving as the proxy can be connected with an IMD programmer through an audio cable. As shown in Figure 6.2, the one end of the audio cable is plugged into the smartphone’s headphone
port while the other end links to an audio-to-USB adapter, which is then plugged into the USB port of the programmer. The audio-to-USB adapter is actually an external sound card with digital-to-analog converter (DAC) and analog-to-digital converter (ADC), hence the analog signal transmitting over the audio cable can be converted into digital data, or vice versa. The access control mobile application is always running in the background, ready to process the incoming requests from the audio cable. The advantages of using the audio cable for communications include:

- **No patient involvement required.** The data can be transmitted through audio cable in a plug-and-play manner, even if the phone is in the lock-up mode.

- **Reduce the attack surface.** The packets exchanged are no longer exposed in the air. The remote adversary cannot overhear or jam the communications.

- **Reduce the energy consumption.** Wireless transmissions consume at least 10 times more power than wired transmissions when providing comparable access rates and traffic volumes [115]. Although both proxy and programmer are assumed to have sufficient power, energy saving becomes a critical concern when the phone battery is low.

- **Proof of proximity.** The programmer operator needs to plug the audio cable into the headphone port of the smartphone, which proves the proximity of the operator to the patient.

- **Low cost.** The audio cable connection does not require extra hardware. An audio-to-USB adapter only costs around $10.

Note that although a few new smartphone models like the iPhone 7 has removed the headphone jack, the audio cable can be connected to the lighting port via a simple adapter. And still, no manual approval is required for communication through an audio cable.
6.3 Attribute-based Encryption

Our scheme achieves attribute-based access control using the attribute-based encryption (ABE) technique. ABE is a one-to-many encryption method, which allows data to be encrypted based on a set of attributes, so that only those users who own the specified attributes are able to correctly decrypt. ABE is collusion-resistant, which can prevent colluding users to gain access by combining their associated attributes, if none of them possesses the full set of required attributes.

There are two major types of ABE schemes: Key-Policy Attribute-Based Encryption (KP-ABE) [116] and Ciphertext-Policy Attribute-Based Encryption (CP-ABE) [117]. In KP-ABE, users’ secret keys are generated based on an access tree (i.e. access policy) whose leaves are associated with attributes, and the data are encrypted over a set of attributes. Since the access policy is embedded in the decryptor’s secret keys, the data encryptor has no control over who can access the data. However, in the IMD access control context, the proxy must check the programmer operator’s qualifications, which are formed as a policy composed of a specific set of attributes. This requirement can be satisfied by CP-ABE, in which the users’ secret keys are generated over a set of attributes and the ciphertext specifies the access policy.

Therefore, we adopt CP-ABE to implement the IMD access control. Specifically, the proxy encrypts the verification message with CP-ABE and sends the ciphertext to the programmer. The programmer can successfully decrypt the message only if the operator’s attributes satisfy the access policy specified in the ciphertext. In contrast, the adversaries cannot decrypt the ciphertext, even if they collude. The CP-ABE scheme consists of the following four fundamental algorithms:

- **Setup**($k$). The Setup algorithm takes a security parameter $k$ as input and randomly picks two exponents, to calculate the public parameters $PK$ and the master key $MK$. $PK$ will be used for encryption, while $MK$ will be used to generate users’
secret keys and is known only to the central authority.

- **Encryption**\((PK, M, T)\). The encryption algorithm takes as input the public parameters \(PK\), a plaintext message \(M\), and an access tree structure \(T\) over the universe of attributes. This algorithm will encrypt \(M\), and produce a ciphertext \(CT\) which only users who possess a set of attributes that satisfies the access structure \(T\) are able to decrypt.

- **Key Generation**\((MK, S)\). The secret key generation algorithm takes as input the set of attributes \(S\) that user \(U\) owns, the master key \(MK\), and randomly selects a set of \(|S| + 1\) numbers which is specific to user \(U\). It outputs a secret key \(SK\).

- **Decrypt**\((PK, CT, SK)\). The decryption algorithm takes as input the public parameters \(PK\), a ciphertext \(CT\) which contains the access policy \(T\), and a secret key \(SK\) generated from attribute set \(S\). If the set \(S\) of attributes satisfies the access policy \(T\), the algorithm will successfully decrypt the ciphertext and return the plaintext message \(M\).

### 6.4 Protocol Design

#### 6.4.1 System Overview

- **IMD.** Each IMD has a unique identification \(ID_i\) and a master key \(K^M_i\). Note that the master key is only used for pairing up the IMD with a proxy device, and will not directly participate the access control procedure.

- **Programmer.** The programmer can be simply viewed as the terminal device used by its operator to interact with the IMD or proxy. It obtains all information required for access control (e.g., secret keys, certificates) from the operator, by manual input or reading in from a smart card.

- **Operator.** The programmer operator is the actual subject to be verified. All
legitimate operators must first be registered at a Central Health Authority (CHA), which manages the qualifications of operators and issues digital certificates for them. Each operator has a unique identification $ID_o$, a pair of public/private keys $KU$ and $KR$, and a public key certificate $Cert$. The qualifications that an operator owns correspond to a set of attributes $S$. CHA will generate the secret key $SK$ for the operator based on the set of attributes $S$. $SK$ can be used to decrypt the ciphertext produced by CP-ABE if the access policy is satisfied. Besides, all operators know the public parameters $PK$ used in CP-ABE.

- **Proxy.** The proxy device has the identification $ID_p$. There is a client program running on the proxy to perform the access control for the IMD. The proxy has been paired up with the IMD through initial setup. The client program has a copy of the public parameters $PK$ used to run CP-ABE, and is able to generate the access tree (policy) $T$ that describes the qualifications required for access.

Although currently there is no such CHA in operation, with the rapid development of e-health and related technologies, it is expected that a CHA (or similar entity) will be available in the near future.

### 6.4.2 Access Control Protocol Design

The access control procedure includes two separate processes: the authentication of the programmer operator and the authorization for access.

Although the IBAC model is not suitable in IMD context and we utilize attributes to control access instead, it is still necessary to authenticate the identity of the programmer operator. One reason for that is to provide non-repudiation guarantee in case of medical disputes. For example, a programmer operator cannot deny his/her access if the start time and end time of the access have been signed by his/her own private key. The other reason is that the ciphertext $CT$ generated by CP-
ABE contains the access policy in plaintext. The access policy specifies the expected qualifications of the authorized physicians (e.g., specialty) and the information related to the IMD model (e.g., the certifications required to operate), which are all very sensitive with regard to the patient’s privacy and should not be publicly accessible to anyone who requests for access. Therefore, our scheme authenticates the programmer operator before the authorization stage, so that only a legitimate operator who has registered at CHA (not necessarily authorized to access) can continue to the authorization process and view the access policy.

In the authorization stage, the proxy encrypts a randomly generated temporary session key $K_t$ with CP-ABE and sends the ciphertext to the programmer. If the programmer operator is an eligible physician whose qualifications (attributes) satisfy the access policy, the session key $K_t$ can be correctly retrieved and used to establish a secured communication channel with the IMD.

The access control procedure is presented in Figure 6.3. We assume that the proxy device has already been paired up with the IMD, and a pair of symmetric keys $K_s$ have
been shared between them for encrypted communications. The detailed procedure is described as follows:

1. The programmer initializes the access control protocol by connecting with the proxy via audio cable, and sending an access request which is composed of a unique action sequence “access.req”, the operator’s digital certificate Cert, a random selected session number SN, timestamp \( t_1 \), and a signature \( \text{Sig}_1 \) signed by the operator’s private key \( KR \). The certificate contains the operator’s public key \( KP \) and identification \( ID_o \). The signature \( \text{Sig}_1 = \text{Sign}_{KR}(ID_o|SN|t_1) \) is attached to prove that the current access requestor is indeed \( ID_o \).

2. The access control mobile application has registered a receiver of the headset connection state changes. When this program is notified of the plug-in event, it will read in and demodulate the audio data. If the action sequence “access.req” can be found in the demodulated data, it indicates that the data is for IMD access request instead of regular audio (e.g., music). Then it extracts and verifies the received \( \text{Sig}_1 \) using the requestor’s certified public key \( KP \) embedded in Cert. If the signature is valid, the programmer operator is successfully authenticated.

3. The proxy will next check if that operator is authorized to access. Specifically, it randomly generates a temporary session key \( K_t \) and encrypts \( K_t \) using CP-ABE. Then, the produced ciphertext \( CT = \hat{E}_{PK,T}(K_t) \) is sent back to the programmer.

4. The programmer decrypts \( CT \) with the programmer operator’s secret key \( SK \) (generated and assigned by the CHA). If the programmer operator’s qualifications (attributes) satisfy the access policy, the temporary session key \( K_t \) will be correctly retrieved. Then, it calculates the hash value of \( K_t \) and sends the hash value \( H(K_t) \) to the proxy.

4. The proxy also calculates the hash value of \( K_t \) with the same hash function. If the two hash values are equal, it indicates that the programmer operator
has successfully decrypted $CT$ and is eligible to access. The proxy will inform the IMD that the programmer $ID_o$ has been authenticated and is authorized to access, and sends a copy of the session key $K_t$ to the IMD. Note that all communications to/from the IMD are conducted in the wireless channel, which may suffer eavesdropping, replay, and tampering attacks. Hence, the session key $K_t$ is encrypted by the pre-shared key $K_s$ to prevent eavesdropping. A timestamp $t_2$ is added to defend replay attacks. The keyed-hash message authentication code (HMAC) of the message is calculated using $K_s$ to ensure the authenticity and integrity of the message.

5. After receiving the authorization notification from the proxy, the IMD retrieves the $K_t$ and sends “ready” message to the programmer.

6. In the mutual communications between the IMD and the programmer, the operation commands $C$ sent by the programmer and the data/result $D$ returned by the IMD are all encrypted using the temporary session key $K_t$. Each authorization permits multiple operations (e.g., data reading, drug delivery). We only draw one round of operation in Figure 6.3 for illustration. The timestamp and HMAC are also adopted in the communications between the IMD and the programmer to defeat various active wireless attacks.

7. After the programmer has completed the access, it sends a “logout” notification message to the proxy. The message includes a signature $Sig_2 = Sign_{KR}(ID_o|SN|t_5)$ in which the access end time $t_5$ is signed.

8. Finally, the proxy will notify the IMD that the current session has ended so that the session key $K_t$ will be removed. Timestamp $t_6$ and $HMAC_4$ are included in the this message.

Our scheme asks the programmer to explicitly log out the session, and requires it to sign the time that the session ends. Therefore, $Sig_1$ and $Sig_2$ together can
prove that the operator has accessed the IMD in that period of time. Any wrong operations performed in this period will be attributed to that specific operator. The programmer should maintain the wired connection with the proxy during interacting with the IMD, to eliminate the possibility that a second programmer is connected with the proxy while the first one is still interacting with the IMD. However, it may happen that the programmer does not sign out by the end of its session for certain reasons, a time-out mechanism is used to tackle this situation. Specifically, during a session, if there is no interaction made by the programmer for a fixed amount of time \( T_{out} \), the session will be closed by the IMD and the session key \( K_t \) will be disabled. Then, the IMD will inform the proxy.

It is an option to use the proxy as a relay for all communications between the IMD and the programmer. In this work, we choose another option (the proxy just assigns the session keys), because the relaying approach causes communications to take longer time, which is a serious problem in case of emergency treatments.

6.5 Security Analysis

6.5.1 Resistance to Passive Attacks

In the wireless channel, the adversary is able to eavesdrop the communications between the proxy and the IMD, as well as between the programmer and the IMD, to obtain the data, operation commands, or keys being transmitted. To defend against such passive wireless attack, all sensitive information is protected with symmetric encryptions. Specifically, a pair of keys \( K_s \) have been pre-shared by IMD and the proxy (during initial setup), and another pair of keys \( K_t \) are securely distributed to IMD and the programmer. The adversary cannot decrypt the ciphertexts without the cryptographic keys.
Over the audio cable, even if the proxy has been infected by malware which will overhear the audio data received by the proxy, our scheme can ensure that it cannot obtain any sensitive information. The temporary session key $K_t$ is encrypted by CP-ABE. The malware cannot correctly derive the plaintexts without the required qualifications (attributes). Additionally, it cannot infer $K_t$ from its hash value $H(K_t)$.

### 6.5.2 Resistance to Active Attacks

In the wireless channel, all messages containing sensitive information are timestamped to prevent replay attacks. In addition, a HMAC is calculated using the corresponding shared keys ($K_s$ between the proxy and the IMD, or $K_t$ between the programmer and the IMD), to defeat the tampering attacks.

Over the audio cable, the malware on the proxy can send arbitrary messages to the programmer or tamper the messages sent by the application running the access control. In our scheme, the only outgoing message to the programmer via the wired audio channel is $CT$. However, either the replay or manipulation of this message cannot help an unauthorized programmer operator to gain access, since only eligible operators can successfully decrypt $CT$ and retrieve the session key $K_t$.

### 6.5.3 Other Attacks

Over the audio cable, the mobile malware can add noise to the audio data sent by the access control mobile application, but this sort of attacks can only disrupt the access of legitimate operators (e.g. causing the failure of authentication or authorization). As we have mentioned before, DoS attacks are not the focus of this work. Besides, our scheme is well-compatible with existing solutions to the proxy-absence spoofing attacks [104] and the resource depletion attacks [100–102].
6.6 Evaluation

In this section, we implement our scheme on real devices and evaluate the overheads of our scheme with experiments.

6.6.1 Prototype Implementation

The key challenge of the implementation of our scheme is the difficulty in obtaining open-source commercial IMD products. Alternatively, in our prototype system, we use TelosB Mote TPR2420 sensor node with 8 MHz TI MSP430 microcontroller, 10kB RAM, and 48kB Flash Memory as the replacement of the IMD. We choose the Rasberry Pi Version 3 Model B, a small single-board device with 1.2GHz 64-bit quad-core CPU and 1GB RAM, to emulate the programmer. A Nexus 4 smartphone powered by a 1.5 GHz quad-core processor with 2 GB of RAM, is used to emulate the proxy.

In our experiments, symmetric encryption and public-key cryptography are implemented using the Advanced Encryption Standard (AES) algorithm and RSA algorithm, respectively. We use 128 bits key for the AES encryption and 1024 bit keys for the RSA encryption. SHA-1 is chosen for the calculation of HMAC.

6.6.2 Testbed Specification

We developed an Android application to delegate the access control on the Nexus 4 smartphone. This mobile program is able to perform CP-ABE and modulate/demodulate the audio signals. Correspondingly, we also developed a client program running on the Raspberry Pi, which can decrypt the ciphertext encrypted with CP-ABE, if the programmer operator’s attributes satisfy the access policy embedded in the ciphertext. This client program is also capable of
modulating/demodulating the audio signals. The smartphone and the programmer is connected via an audio cable and a SYBA external USB Sound Adapter (audio-to-USB convertor). A screenshot of the prototype system is shown in Figure 6.4.

We adopted the Binary Frequency-Shift Keying (BFSK) frequency modulation scheme for modulation, in which the digital data are converted into the analog signals at two different frequencies for transmission over the audio cable. For example, the binary “0” bit is represented by the audio signal at frequency $f_0$ while the binary “1” bit is represented by the audio signal at frequency $f_1$. Figure 6.5 displays the results of the signal modulation and the spectrum analysis. The red signal (upper sine waves) in Figure 6.5 is the analog signal after modulated by BFSK. At the receiver end, the analog signal is sampled to a sequence of discrete-time signal (samples). Then, we use the Discrete Fourier Transform (DFT) algorithm to convert the sampled analog signal from time domain into the frequency domain representation, which is illustrated in Figure 6.5 as the blue pulse-like signal. Specifically, the analog signal to be demodulated can be viewed as an addition of multiple sine signals in different frequencies. With Fourier transform, the magnitudes of the modulated signal on various frequencies within the spectrum range are calculated. The frequency with
the highest amplitude (i.e., maximum power) is called the peak frequency. If the peak frequency equals $f_0$, then the current signal sample represents a “0” bit; while if the peak frequency equals $f_1$, then the signal sample represents a “1” bit. The parameters we used for modulation and demodulation are listed in Table 6.1.

### 6.6.3 Experimental Results

To evaluate the efficiency of our scheme, we measure the computational overheads of the protocols running on the IMD (TelosB Mote), the proxy (Nexus 4 smartphone), and the IMD programmer (Rasberry Pi), respectively. All the run-time overheads are the average of 50 measurements.
6.6.3.1 IMD

Our scheme uses only the basic operations and computations in the IMD to ensure security (i.e., symmetric encryption and HMAC). On TelosB node, the 128-bit AES encryption takes 2ms. For HMAC, we estimate the length of the plaintext message (“operate, rep”, $SN, ID_i, ID_o, E_{K_t(D)}$, $t_4$) to be 78 bytes in total, including a 4-byte command, a 2-byte sequence number, two 4-byte IDs, and a 64-byte data/result returned). The HMAC computation over a 78-byte message takes 47ms.

6.6.3.2 IMD programmer and proxy

The IMD programmer (Raspberry Pi) and the proxy (smartphone) both need to conduct modulation/demodulation for the wired communication through the audio cable. Figure 6.6 and Figure 6.7 show the time consumptions for modulation and demodulation over different data sizes on the smartphone and Raspberry Pi, respectively. As we can see, the smartphone has a better performance than the Raspberry Pi, and the demodulation takes longer time than the modulation process.
Additionally, the proxy (smartphone) needs to encrypt the temporary session key with CP-ABE algorithm, and the IMD programmer (Raspberry Pi) needs to run the decryption algorithm to retrieve the session key. Figure 6.8 and Figure 6.9 show the time consumptions for the CP-ABE encryption on smartphone and decryption on Raspberry Pi, respectively. The run-time overheads for CP-ABE encryption and decryption both increase with the number of leaf nodes (attributes). Our experiments tested a maximum of 20 attributes, which are sufficient to specify the qualifications of the programmer operator. The decryption is found to be the most time-consuming step in the protocol, which takes about twice the time used for encryption. Since the CP-ABE encryption/decryption is required only once, their execution time are considered acceptable.

6.7 Literature Review

Most existing works employ the two-party access control architecture, in which the IMD with constrained computational power, battery capacity and storage must
perform the access control by itself, resulting in a relatively weak security and shorter IMD lifetime. Some works proposed that the IMD should be accessible only to a group of trusted people (e.g., doctors, relatives) who have been added into the IMD’s access control list (ACL) [118] [119]. However, the limited storage of the IMD will greatly restrict the scalability of such schemes, considering the number of eligible physicians is very large. In addition, the authentication of the requestor requires the verification of the certificate/signature. The computational power and battery capacity of the IMD can hardly afford the public-key cryptography.

Some other two-party based access control schemes only check whether the programmer is in proximity. Specifically, the IMD and programmer need to extract certain features from the same source simultaneously, and generate a temporary key based on the extracted features, respectively. The source is usually the signal in an out-of-band channel, such as electrocardiography (ECG) signal [106], body-coupled electric signal [107], vibration [108], ultrasound [109], etc. If the programmer is close enough to the IMD, they will derive the same temporary key for the symmetric encryption of their communications. However, the real-time signal measurement and key extraction computations both bring in an extra burden to the resource-constrained IMD. Other works proposed to pre-load the patient’s biometric information (e.g., fingerprint, iris) or a password into the IMD. During access, the programmer operator can manually collect the biometric features from the patient [112] or the password from a physical object carried by the patient [113]. However, the bio-features and the password engraved on an object could be inadvertently disclosed to an adversary. Besides, a common disadvantage of these two types of schemes is that they only provide a coarse-grained access control. No information about the access requestor is acquired and validated.

By contrast, using the proxy-based access control architecture, the proxy device can handle those resource-consuming tasks. It can support more complicated and
robust access control schemes. However, previous proxy-based access control schemes either depend on anomaly detection of the access pattern [102, 105] or only verify the authenticity of the programmer [103, 104]. The former category of methods do not check the access requestor at all, and require a training process that is specific to each patient. The second category of methods is vulnerable considering that an attacker could also purchase or steal a legitimate programmer. Our scheme views the programmer and its operator together as one subject, and the access decision is made based on the eligibility of the programmer operator. We not only authenticate the operator, but also employ the CP-ABE algorithm to verify the qualifications of the operator.

6.8 Chapter Summary

In this chapter, we propose a novel fine-grained IMD access control scheme assisted by a proxy device such as the patient’s smartphone, which will delegate the heavy access control computations for the IMDs. To mitigate the potential wireless attacks, the communications between the proxy and the IMD programmer are conducted through an audio cable. The ciphertext-policy attribute-based encryption is employed to enforce the fine-grained access control over the qualifications of the programmer operator. We build a prototype system to evaluate our scheme. The experimental results demonstrate its feasibility and effectiveness.

Our major contributions can be summarized as follows:

- We first comprehensively study and analyze various existing IMD access control schemes in terms of the access control architecture and access control model. We also take a full consideration of the special use cases that a good IMD control scheme should be able to handle.

- Our proxy-based IMD access control scheme can greatly alleviate the
computational overhead and power consumption of IMDs. The proxy communicates with the IMD programmer via the audio cable, which can defend against the wireless passive and active attacks. Unlike USB connection, communications through headphone jack does not require the patient to unlock the device for manual approval, which is a practical concern especially for patients who are unconscious.

- Our scheme adopts the ciphertext-policy attribute-based encryption (CP-ABE) to provide a fine-grained access control over the qualifications of the programmer operator. Specifically, the proxy encrypts the temporary session key with a specific access policy and sends the ciphertext to the IMD programmer. If the programmer operator owns the set of required attributes (qualifications), the temporary session key can be correctly retrieved.

- We implement our scheme on real emulator devices: the IMD is emulated by TelosB mote, the proxy is emulated by smartphone, and the IMD programmer is emulated by Rasberry Pi 3. The evaluation results show that the proposed scheme is lightweight and effective.
CHAPTER 7

DISSETATION CONCLUSION

In this dissertation, we conduct an in-depth study of the security and privacy issues in modern wireless and mobile devices, including smartphones, IoT devices and IMDs. Although these devices have been very popular on market and have a rapidly growing number of users, their security vulnerabilities keep being discovered. Both the academia and industry have been working closely in fixing these flaws and bugs, and devising new security mechanisms.

We first investigate the phishing attacks, clickjacking attacks, and camera-based attacks targeted on smartphones. The corresponding countermeasures are proposed to defend against these attacks. Then, motivated by the limitations of existing works, we design two novel access control schemes for IoT devices and IMDs, respectively.

7.1 Summary of Contributions

Smartphones are not secured at the same level as PCs, due to the differences in both the operating system (software) and hardware. Specifically, many existing security schemes on PCs cannot be migrated to the mobile platform, leaving the mobile systems not well-protected. User interface attacks are the typical examples that have become particularly dangerous on smartphones due to the limitations of the computational power and the phone screen size.

Phishing attacks aim to steal private information and credentials by way of impersonating a legitimate entity. However, mobile browsers remove the status bar,
and hide the URL bar (truncated) once a web page finishes loading, which gives the phishing webpages excellent chance to spoof the user. Meanwhile, conventional black-list based detection adopted by PC browsers is disabled on mobile browsers, and the HTML source code based detection is found to be not reliable. We propose MobiFish, a novel lightweight mobile anti-phishing scheme to defend against phishing webpages and applications, without the reliance on the HTML source code. The key idea is to check the consistency between the claimed identity and the actual identity. The actual identity of a webpage or app is obtained from the secondary-level domain name of the server it tries to connect, while the claimed identity is displayed in the UI presented to the user. Our detection scheme checks the existence of identity discrepancy by extracting all texts from the screenshot of the login UI using optical character recognition technique, and searching for the actual identity among the extracted texts.

Clickjacking is another type of UI attacks, which inserts an opaque and untouchable layer on top of the screen (attack window), and tricks the user to click on a specific position for malicious purposes. The click event seemingly going to the top front attack window actually goes to the victim window underneath. Since the conventional clickjacking attacks on PCs target at webpages and there have already been tons of solutions, we mainly focus on the mobile clickjacking attacks on mobile application UIs and system UIs. The unique features of clickjacking are analyzed, and the “after-attack” disguises to keep the clickjacking undiscovered is investigated. Specifically, three side-channels are presented which allow the malicious app to listen to the user input events so that the attack window can react instantly. To better evaluate the threats, we explored a variety of clickjacking attacks on system apps, 3rd-party apps, and other particular system UIs. Considering the limitations of existing approaches, we propose a novel effective defense scheme is proposed and added into the Android system as an independent system service. Our solution is lightweight and automatic,
which requires no user/developer effort and is compatible with existing apps.

We have also studied the spy camera attacks on smartphones, which can severely violate user privacy by stealthily taking videos or photos. We developed two advanced camera based attacks: the first one allows the remote attacker to control and monitor through camera view in real time; the second attack records the user’s eye movements when entering password/PIN given the assumption that the user’s eyes may move along with the keys being touched. We demonstrate that it is possible to recover simple passwords like the PIN through an offline processing of the video containing the eye movements. Finally, we propose an effective detection scheme based on the specific launch patterns of the spy camera attacks.

Besides, we design two novel access control schemes for IoT devices and IMDs, respectively.

To secure the home IoT devices from being accessed by external malicious device, we propose a secondary authentication scheme over an out-of-band channel. Specifically, all legitimate home IoT devices are registered on Blockchain. They can help each other to verify if the access requestor is inside the home or not. For example, the verifier device sends a sequence of code encoded in light or audio signals, which only devices within the home can receive and correctly decode. The Blockchain is used to record the device relationship information and the secondary verification result.

The access control for implantable medical devices (IMD) is critical since it is closely related to the patient’s treatment and well-being. To secure the wireless interface of the IMD, we propose to pair it with the patient’s smartphone so that the phone can authenticate the IMD programmer on behalf of the IMD. During access authorization, the programmer operator has to be physically present near the patient and connect the programmer to the user’s phone through an audio cable. CP-ABE algorithm is adopted to enforce a fine-grained access control over the qualifications.
of the operator. After the programmer is authenticated and authorized to access, a pair of symmetric keys will be issued to the IMD and the programmer for secure communications. Our scheme can work even if the user is in an emergency condition (e.g., unconscious) and if the phone screen is locked.

### 7.2 Future Research

The security and privacy issues of modern wireless and mobile devices will continue to raise as a hot topic. In the future research, we plan to still focus on the security of smartphones/tablets, wearable devices, IoT devices, and IMDs.

For smartphone security, we have investigated the major mobile UI deception attacks (i.e., phishing and clickjacking). We can use these experience on the detection of other mobile spoofing attacks, such as malicious in-app advertisements, fraud text messages and emails.

For the security of IoT devices and IMDs, we aim to devise common security standards that can be applied to all different types of IoT devices and IMDs, including the authentication, authorization, communication security, etc.

We will seek to adopt the machine learning and other techniques used in artificial intelligence into our security mechanisms. We will also study the viability of using more biometric factor and ambient context information for security enhancement.
PUBLICATIONS

Journal Publications


Conference Publications


Communications and Network Security (CNS), 2016.


Submitted


BIBLIOGRAPHY
BIBLIOGRAPHY


medical devices with the external wearable guardian,” in *IEEE INFOCOM*, 2011.


