DISS. ETH No. 24485

Security of User Interfaces: Attacks and Countermeasures

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

LUKA MALIŠA

Master of Science in Computer Science,
ETH Zurich, Switzerland

born on 31.12.1984

citizen of Croatia

accepted on the recommendation of

Prof. Dr. Srdjan Čapkun, examiner
Prof. Dr. Kevin R. B. Butler, co-examiner
Prof. Dr. William Enck, co-examiner
Prof. Dr. Apu Kapadia, co-examiner
Prof. Dr. Adrian Perrig, co-examiner

2017

Come to the edge.
We might fall.
Come to the edge.
It's too high!
COME TO THE EDGE!
And they came,
And he pushed,
And they flew.

— Christopher Logue

Dedicated to parents
For helping us grow

Abstract

User interfaces (UIs) are the means through which we interact with computer systems, and users perform both simple, as well as critical task through such user interfaces. For example, users visit their daily news portals, but also perform e-banking payments through user interfaces. Medical doctors use them to operate safety-critical devices such as respirators, implanted medical device programmers, etc. Given that safety- and security-critical tasks are performed through such user interfaces, it is important to secure them against attacks. Therefore, the goal of this thesis is to (1) better understand the security problems of modern user interfaces, and (2) propose novel defenses against damaging user interface attacks.

There is a plethora of known user interface attack approaches that launch attacks from, e.g., a malicious application running on the target device, or from malicious peripherals (e.g., a mouse or a keyboard). Such attacks can, for example, infer user input or inject malicious input into the system. However, they commonly suffer from accuracy issues or limited attack applicability. Different systems for detecting user interface attacks were also proposed. However, they are commonly vulnerable to evasion through simple obfuscation attacks.

In this thesis, we address these shortcomings and make the following contributions. First, we propose two new user interface attacks that are accurate, hard to detect, and enable previously unreachable attack scenarios. Second, we propose two new systems for detecting a particularly damaging and effective user interface attack — phishing. Our systems are based on visual similarity and are resilient to obfuscation.

Zusammenfassung

Benutzerschnittstellen (UIs) erlauben es uns mit Computersystemen zu interagieren. Sie ermöglichen es Benutzern sowohl einfache als auch kritische Aufgaben auszuführen. Beispiele für Benutzerschnittstellen sind Nachrichtenportale, auf denen Benutzer täglich Neuigkeiten abrufen, oder e-banking Plattformen, über die sie ihr Zahlungen online abwickeln können. Mediziner brauchen Benutzerschnittstellen, um sicherheitskritische, medizinische Instrumente wie Beatmungsgeräte zu bedienen oder um implantierte Geräte zu programmieren. Da sicherheitskritische Anwendungen über Benutzerschnittstellen ausgeführt werden, ist es wichtig diese gegen Angriffe abzusichern. Das Ziel dieser Doktorarbeit ist es (1) Sicherheitsprobleme moderner Benutzerschnittstellen besser zu verstehen und (2) neue Verteidigungsmechanismen gegen Benutzerschnittellenangriffe zu finden.

Es gibt eine Unmenge an bekannten Benutzerschnittstellenangriffen, die mit Hilfe von schädlichen Anwendungen, welche auf dem Zielgerät installiert sind, oder die durch schädliche Eingabegeräte (z.B. eine Computermaus oder -tastatur) durchgeführt werden. Solche Attacken können unter anderem Rückschlüsse auf Benutzereingaben machen oder schädliche Eingaben in dem betroffenen System ausführen. Diesen Angriffen mangelt es jedoch an Präzession und sie verfügen nur über eine beschränkte Anwendbarkeit. Es wurden bereits verschiedene Systeme zum Aufdecken solcher Schnittstellenangriffe vorgeschlagen. Diese können jedoch häufig mit einfachen Verschleierungstaktiken umgangen werden.

In dieser Doktorarbeit beschreiben wir die jeweiligen Schwächen und leisten den folgenden Forschungsbeitrag: Erstens stellen wir zwei neue Benutzerschnittstellenangriffe vor, welche präzise und schwer zu entdecken sind, sowie bisher unerreichbare Angriffsszenarien ermöglichen. Zweitens stellen wir zwei neue Systeme vor, die einen besonders schädigenden und effektiven Benutzerschnittstellenangriff entdecken können, der “Phishing” genannt wird. Unser System beruht auf visueller Ähnlichkeit und funk-

tioniert auch dann, wenn Verschleierungstaktiken angewendet werden.

Acknowledgments

Reaching the end of this journey would not have been possible without the two special people that enabled me to embark upon it to begin with. I thank my advisor Prof. Dr. Srdjan Čapkun for his guidance, his never-ending support, and his seemingly infinite amount of patience. He watched me stumble and fall many times while navigating the scientific waters, yet he never lost faith in me and always helped me learn and grow, both on a personal and on an academic level. I also thank Prof. Dr. Mario Čagalj for introducing me to the idea of a PhD abroad all those years ago.

I would like to thank the members of my committee, namely Prof. Dr. Kevin R. B. Butler, Prof. Dr. William Enck, Prof. Dr. Apu Kapadia, and Prof. Dr. Adrian Perrig for dedicating their time and for providing invaluable insight and dissertation comments.

The beginnings of my PhD were a particularly challenging period. I would therefore like to thank Prof. Dr. Nils Ole Tippenhauer, Prof. Dr. Kasper Rasmussen, Dr. Claudio Soriente, Prof. Dr. Aurélien Francillon, and Dr. Boris Danev for their help and guidance during those initial times. I sincerely and wholeheartedly thank both Mrs. Barbara Pfändner and Mrs. Denise Spicher for helping me innumerable times over the years.

During my studies at ETH, I had the privilege and honor of interacting and befriending many great people. I thank my colleagues Prof. Dr. Aanjhan Ranganathan, Aritra Dhar, Dr. Alexandra Dmitrienko, Prof. Dr. Arthur Gervais, Marco Guarnieri, Nikolaos Karapanos, Dr. Kari Kostiainen, Dr. Claudio Marforio, Dr. Ramya Masti, Dr. Ognjen Marić, Siniša Matetić, Daniel Moser, Mridula Singh, Hubert Ritzdorf, Prof. Dr. Joel Reardon, David Sommer, Dr. Elizabeth Stobert, Der-Yeuan Yu, and Thilo Weghorn for making my PhD a truly memorable experience. I thank Stephanos "Stipe" Matsumoto and Thilo Weghorn for all the camaraderie and the many long and illuminating conversations directed towards spiritual growth.

I especially thank Aanjhan Ranganathan for the psychological counseling services rendered during the years. He patiently listened to all my thoughts, fears and complaints, and all it cost me were a few dozen cups of coffee. Your words of encouragement and wisdom, your empathy and general understanding were much appreciated and will never be forgotten.

I extend special thanks to my co-authors Michael Och and Thomas Knell, who contributed significantly towards their respective projects and thereby helped shape this dissertation into what it is today.

I would like to dedicate a special part of this acknowledgment to Kari Kostiainen. Without his guidance, his patience and his clarity of thought, I would have given up on my PhD. Thank you!

Finally, I would like to thank my mother, as well as Ivana Karninčić for all the love and support, and for standing by me during good and bad.

Completing a PhD at ETH Zurich was a challenging, but also immensely rewarding endeavor. However, as with all things, and a PhD is no exception, they eventually come to an end.

Therefore, onwards to new challenges, whatever they may be!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A user interface (or "UI" for short) is the means by which we interact with a computer system, and any user interface comprises of two main components: the user input, and the system output mechanisms. The technology of user interfaces has progressed far, from the early days of systems based on bulky punch cards and command line interfaces. However, even though the technology has steadily evolved over the years, the core underlying principle of our interaction with computers stayed surprisingly unchanged in the past 40 years. The most prevalent type of user interfaces today are still those tailored for the combination of a pointing device (touch sensor or mouse), and a visual display. They are present in the majority of modern devices, such as laptops and desktop computers, as well as on mobile platforms such as smartphones and tablets. Securing such user interfaces is the focus of this thesis.

Users perform both simple daily tasks as well as operations critical to their privacy, security, and safety through user interfaces. For example, users visit news portals and check weather forecasts, but also perform administrative system configurations, write e-mails, or enter their credit card information and e-banking credentials. Medical doctors use such UIs to configure the implanted medical device of a patient, while assembly-line operators use them to control robots.

Users operate a variety of input (e.g., keyboard, touch-screen, mouse) and output peripheral types (video screens, speakers, etc.), with different connection interfaces (e.g, USB, FireWire, Thunderbolt). The complexity and diversity of modern UIs present an inviting attack surface, resulting in an entire class of attacks called *user interface attacks*.

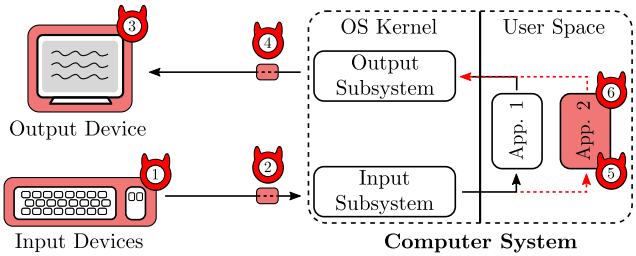


Figure 1.1: *An overview of attack vectors on user interfaces. Through physical alterations to the target system, the adversary can compromise both the confidentiality and integrity of user input (malicious peripherals (1), man-in-the-middle devices (2)), as well as system output (compromised interconnecting link (3), or output device (4)). A malicious application can also compromise confidentiality of user input (input-inference attacks (5)), as well as integrity of output (e.g., spoofing attacks (6)). Attacks where the adversary compromises the OS are outside of the scope of this thesis.*

An adversary can launch UI attacks, that can compromise the *confidentiality* and *integrity* of both user input and system output, in different ways (Figure 1.1). Some attack vectors require physical access, while others only require a malicious application running on the target system. For example, an adversary with physical access (1, 3 in Figure 1.1) can install malicious input peripherals [84, 99], or a malicious device located in-between the legitimate peripherals and the system [99] (2, 4). The adversary can then compromise the confidentiality of all user input (e.g., by using key-loggers [98]), but can also perform more sophisticated attacks that compromise the integrity of user input by injecting malicious UI commands [84, 99, 123] (e.g., opening an administrative console and executing a malicious application). Similarly, the adversary can compromise the confidentiality and integrity of system output by, e.g., directly compromising the output device, or by installing similar man-in-the-middle attack devices. The peripheral ports of a device are often directly accessible [119], allowing an adversary to easily attach a stealthy and malicious hardware device.

If the adversary controls a malicious application running in the background of a smartphone (5, 6), the adversary can also compromise the confidentiality of user input by performing input inference attacks using side channels (e.g., accelerometer [36, 142], gyroscope [134, 196]). Launching such attacks requires few [44, 137, 161] or no special privileges [36, 142].

An entire subclass of user interface attacks focuses on user deception [42, 63, 90, 116, 139, 150]. For example, if the user interface presents content generated by the adversary, the integrity of what the user sees is violated, and the user can be deceived into believing that the target system (or application) is in a different state, and take actions that the user did not intend to perform. Such unintended actions can result in severe safety and security violations, depending on the role of the system itself. For example, if the user interface of an implant programmer shows malicious electrocardiogram data (ECG), a doctor could wrongly conclude that the patient requires immediate electroshock therapy. In phishing attacks, which are another kind of UI attacks, a malicious application steals login credentials by presenting the user with a fake login screen of an application the user knows. Such application-based UI attacks are widespread, as research [214] showed that 86% of analyzed mobile application malware samples were repackaged versions of legitimate apps, which performed UI attacks.

Alternative UI attacks are possible, such as mounting remote software attacks, using side-channels to infer physical keyboard input [110] and video output [102], or compromising the operating system. However, they are outside of the scope of this thesis as such attacks are highly specialized and are significantly less frequent.

User interface attacks are powerful because (1) in case of malware running on the device, they commonly require few to no special permissions to launch, (2) in case of physical attacks, they require only brief and noninvasive access to the target system (an attacker only connects a peripheral), and most importantly, (3) they can bypass existing, commonly deployed, security mechanisms. For example, in the scenario where malware is running on the target system, even though the malware may be unable to perform a malicious operation directly (e.g., initiate electroshocks, read user login credentials), due to lack of permissions, the malware can still *fool the user* into performing the operation instead. In such cases, the system is still vulnerable to compromise, irrespective of any applied architectural security features (system hardening, process isolation, exploit mitigation techniques, or other security mechanisms).

The enabling fact for user interface attacks is the lack of a secure end-to-end channel between the user and the application they are interacting with. However, depending on the considered system, establishing such a channel is challenging. Authenticated (and confidential) input and output through dedicated devices [127], software attestation of peripherals [106], or security indicators [31, 103, 121, 162] are possible solutions. However, such mechanisms only protect against some kinds of UI attacks and are not

widely deployed on modern commodity devices as they commonly require specialized and expensive hardware, or reduce system usability by, e.g., increasing the cognitive load of users.

Instead of establishing such a secure channel, approaches based on preventing or detecting malicious behavior were also proposed. For example, a system can be hardened against malicious peripherals [133], however, such approaches can reduce overall system usability, as in daily usage scenarios users require the ability to connect arbitrary peripheral devices. Even more alerting is that users plug in untrusted USB devices, as 45% to 98% of users connected random and untrusted USB devices found on the street [182]. Access-control approaches for USB peripherals [19, 180] (e.g., USB firewalls) only protect against certain types of input injection attacks (e.g., execute pre-programmed commands through malicious peripherals). Approaches based on extracting and comparing static [65, 215] (e.g., application code) and dynamic [21, 75, 147, 184, 198] (e.g., application system call patterns) fingerprints are susceptible to evasion by means of obfuscation. An adversary can easily evade detection by slightly modifying a malicious application such that it results in different fingerprints [120].

Due to the variety of attack types, and affected platforms (from smartphones and desktop machines, to embedded terminals), user interface attacks are challenging to defend against. Although various countermeasures were already proposed, user interface attacks still remain widespread, and highly successful attack vectors.

1.1 Contributions

The goal of this thesis is to better understand the security problems of modern user interfaces. More precisely, our goals are to (1) better understand the attack surface on modern user interfaces, and (2) to present new ways of preventing widespread types of user interface attacks, that overcome the drawbacks of existing approaches.

Towards that goal, we make the following contributions. We describe two new user input attacks, namely an input inference attack on Android smartphones, and a new class of command injection attacks. We present two new systems for detecting smartphone spoofing attacks.

1.1.1 Attacks

Modern user interfaces are complex, and it is therefore challenging to foresee the effects that the introduction of a new user input method will have on overall system security. *Hover*, or floating touch, is such a new

kind of user input technology that produces a special type of event (hover event). Such events allow the user to interact with the device without physically touching its screen. The hover technology gained popularity when Samsung, a prominent mobile device vendor, adopted it in its Galaxy and Note series. We show how introducing such a novel user input technology can have unexpected, and far-reaching security ramifications, that result in serious violation of user input confidentiality. More specifically, we make the following contributions:

Inferring user input with Hover. We demonstrate a novel input inference attack on modern Android smartphone devices that support the hover technology. Our attack uses such hover events to infer user input, i.e., screen tap locations, or text inserted through the on-device keyboard, and potentially affects millions of users [29, 30, 70, 92]. Existing input inference attacks either require additional permissions [44, 137], depend on environmental factors [137] (e.g., noise in the phone's vicinity), infer only a particular kind of input (e.g., numerical input), or exhibit low inference accuracy. We show that a malicious application, running in the background of an Android device, can infer user input from all applications on the device in a precise and continuous manner. We also demonstrate that our attack can be implemented *without any special permissions*.

However, inferring user input is the initial step, and to derive useful information, the malware needs to know the context (system state) in which it was entered (e.g., whether the user is interacting with a game or an on-screen keyboard). In input inference attacks, the malware is a malicious application running on the device, that can infer the context with the proper permissions, or by using known side-channels (e.g., `/proc`).

In command injection attacks, the malware is a malicious hardware device (e.g., a USB peripheral), and inferring system state is a challenging task, as such malware is not executed on the target devices directly and therefore does not have access to any form of system output. This inability to infer system state is a major limitation of existing command injection attacks. For example, if an adversary connects a malicious peripheral that masquerades itself as a keyboard, the malware cannot know in the current system state and hence cannot determine when to launch its attack. The malware therefore commonly launches the attack immediately upon connection to the system. However, if the malware launches the attack at the wrong point in time, the user may notice, and the attack may fail. Furthermore, we observed that more damaging UI attacks can be performed

while a legitimate user is operating the device (e.g., a doctor programming the implanted pacemaker of a patient).

Adaptive user interface attacks. We present a new class of *adaptive* user interface attacks that overcome those limitations, where the adversary attaches a small device in between the legitimate input peripheral and the target system. By analyzing the stream of user input, our approach infers the most likely state of the system without observing any feedback from the system. Our approach therefore operates "in the blind", and enables launching of precise and damaging UI attacks. Existing command injection attacks only perform simple command injection, e.g., where the malicious device attempts to install malware on the system by executing a pre-programmed sequence of input commands that, e.g., download and execute a malicious application immediately upon connecting the device to the target system. Such attacks are limited because the installed malware, or introduced system misconfigurations, can be detected by existing malware detection approaches.

Our approach enables new attack scenarios that existing command injection attacks could not achieve. Our attack can compromise the integrity of a user's e-banking session without ever resorting to installing malware. We show that such attacks can be implemented efficiently, are hard for the users to detect, and can lead to serious violations of input integrity.

1.1.2 Countermeasures

A common assumption in malware detection approaches is that malware exhibits some form of behavior (e.g., reading user contacts or SMS), or requirement (e.g., special permissions) that a benign application does not, which enables it to be detected. However, compared to regular mobile malware, spoofing applications can be significantly stealthier, as such malware does not require any special permissions, nor does it necessarily perform any suspicious actions, other than drawing on the device screen.

Therefore, applying existing malware detection approaches to phishing apps is challenging. Existing detection approaches based on extracting and comparing static or dynamic fingerprints were proposed. However, we demonstrate that such approaches are susceptible to simple obfuscation attacks [120]. User-assisted approaches (e.g., security indicators) require attentive users, and prior works have shown that users typically ignore or misunderstand such security indicators [58, 160]. To detect spoofing

applications in a manner that is accurate, resilient to obfuscation, and performant, a new approach is needed.

Instead of analyzing a large malware corpora and extracting common denominating similarity features from them, we take a conceptually different approach — we focus on *visual similarity* as perceived by the users. Security-critical user interfaces (e.g., login screens) of mobile applications are commonly significantly simpler in both design and the number of elements, when compared to similar UIs on, e.g., desktop devices. This observation enables us to use various image analysis techniques. Our approach extracts screenshots at runtime, and compares them to known reference values. Furthermore, our approach is more robust to obfuscation as we base our analysis on the end result of every spoofing attack — namely the screenshot presented to the user. More specifically, we make the following contributions:

On-device spoofing detection system. We propose a novel, on-device spoofing detection approach, tailored to the protection of mobile app login screens. We propose *deception rate* as a novel similarity metric for measuring how likely the user is to consider a potential spoofing app as one of the protected applications. We conducted a user study that provided us with insight into how users perceive visual change. Some visual modifications (e.g., reordering elements) the users did not perceive as important as, e.g., modifying the logo. We also used the study results to implement a spoofing detection system based on visual similarity. We show that efficient detection is possible, with low performance overhead.

Even though such an approach is resistant to evading detection by means of obfuscation, it still runs on the user's phone. It therefore represents a *reactive* approach, as it cannot prevent malware infection from taking place, but can only detect the problem once the malware, running on the user's smartphone, launches the attack. We therefore asked the following question: "*Can the approach of visual similarity be used to detect malware on the marketplace as well?*".

Cloud-based spoofing detection system. Motivated by our findings above, we present a novel, and *proactive* approach, for detecting mobile application impersonation attacks on the marketplace. Our system uses dynamic code analysis to extract user interfaces from mobile apps and analyzes the extracted screenshots to detect impersonation. As the detection is based on the visual appearance of the application, as seen by the user, our approach is robust to different attack implementation techniques and

resilient to simple detection avoidance methods such as code obfuscation. Our work demonstrates that impersonation detection through user interface extraction is effective and practical at the scale of a whole marketplace.

1.2 Thesis Organization

This thesis is divided into two parts, and is organized as follows. We begin by motivating the problem of user interface security and reviewing related work in **Chapter 2**.

In the first part of this thesis we propose two novel attacks. In **Chapter 3**, we present an user interface attack on Android, that infers user input in an accurate and continuous manner. Our attack uses a novel side-channel based on the Android hover technology. In **Chapter 4**, we present a new class of user interface attacks that focus on injecting malicious user input, and that enable attack scenarios that existing attacks are not applicable to.

In the second part, we consider the problem of spoofing, a widespread user interface attack. We propose two novel systems for detecting such mobile application spoofing attacks. In **Chapter 5**, we propose an on-device detection system, while in **Chapter 6** we focus a scalable, cloud-based spoofing detection system.

We conclude this thesis in **Chapter 7** by summarizing our findings. Furthermore, we reflect on the lessons learned, and we describe possible future directions in the field of user interface security.

1.3 Publications

The chapters in this thesis correspond to the contributions from the following publications:

- E. Ulqinaku, L. Mališa, J. Stefa, A. Mei, and S. Čapkun, **Using Hover to Compromise the Confidentiality of User Input on Android**, *Proc. ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2017
- L. Mališa, K. Kostiainen, T. Knell, D. Sommer, and S. Čapkun, **Hacking in the Blind: (Almost) Invisible Runtime User Interface Attacks**, *Proc. Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2017
- L. Mališa, K. Kostiainen, and S. Čapkun, **Detecting Mobile Application Spoofing Attacks by Leveraging User Visual Similarity Per-**

ception, *Proc. Conference on Data and Application Security and Privacy (CODASPY)*, 2017

- L. Mališa, K. Kostiainen, M. Och, and S. Čapkun **Mobile Application Impersonation Detection Using Dynamic User Interface Extraction**, *Proc. European Symposium on Research in Computer Security (ESORICS)*, 2016

In addition to the core publications, during my PhD I also co-authored the following papers:

- N. O. Tippenhauer, L. Mališa, A. Ranganathan, and S. Čapkun, **On Limitations of Friendly Jamming for Confidentiality**, *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2013
- E. Androulaki, C. Soriente, L. Mališa, and S. Čapkun, **Enforcing Location and Time-Based Access Control on Cloud-Stored Data**, *Proc. International Conference on Distributed Computing Systems (ICDCS)*, 2014
- D. Sommer, A. Dhar, L. Mališa, E. Mohammadi, D. Ronzani, and S. Čapkun, **CoverUp: Privacy Through "Forced" Participation in Anonymous Communication Networks**, *eprint 2017/191*, 2017
- E. Stobert, E. Cavar, L. Mališa, and D. Sommer, **Teaching Authentication in High Schools: Challenges and Lessons Learned**, *USENIX Workshop on Advances in Security Education (ASE)*, 2017

Chapter 2

Security of Modern User Interfaces

2.1 Introduction

Computer systems are an integral and inseparable part of human society, as they have proved their usefulness in most aspects of our lives. However, a computer system designed to be operated by a user is only useful when accompanied by a corresponding, and useful, *user interface*.

Such user interfaces (or "UI"s for short) consist of two main components: a user input, and system output mechanisms, illustrated in Figure 2.1. The input mechanisms could be the common combination of a pointing device and keyboard, but also other input peripherals, such as touch-screens, joysticks, or microphones. The most common output mechanism is a video screen, but can also be a set of speakers, e.g., in case of audio controlled user interfaces, such as Google Home [78] or Amazon Echo [14].

User interfaces as we know them today changed significantly over the past 50 years, as some of the first user interfaces were based on batch processing systems. Such UIs did not enable real-time interaction with the computer system, as all user input needed to be specified ahead of time (e.g., in the form of punch cards, or magnetic tape), provided to the computer's input queue, and all system output was then produced in the form of printouts. The large delay between providing input and observing system output made user interaction slow and error-prone. A user could

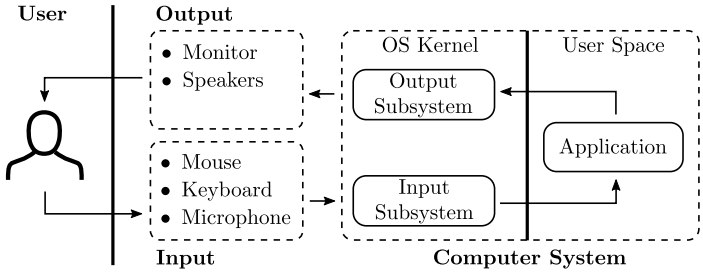


Figure 2.1: *An example structure of modern graphical user interfaces. However, some operating systems (e.g., microkernel designs [87]) can have the input and output subsystems located in user space, instead of the privileged kernel space.*

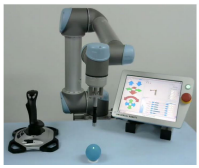
notice an error only after the printout started to occur. The user then had to correct the error, and repeat the input process from the beginning.

The next advancement were command-line interfaces, that provided users with immediate feedback on video displays. Furthermore, such systems enabled users to specify input through keyboards, rather than physical mediums, which allowed users to notice and correct errors immediately.

Our present, de-facto standard, way of interacting with computer systems are graphical user interfaces, and the most widespread graphical UIs are ones based on the *windows, icons, menus, and pointer* model (also known as "WIMP"). Such UIs allow multiple windows, showing different information, to be simultaneously present on the device screen. Menus are used to specify common user commands (e.g., opening or saving files, closing applications, etc.). Pointing, commonly through the use of a mouse or similar device, is an intuitive way to navigate such two-dimensional user interfaces. Contemporary operating systems, such as Windows, OS X, as well as the GNOME [5] and KDE [6] desktop environments, all follow a similar paradigm.

In such a model, UI operations are often metaphors for real-world operations [73] (a concept known as *skeuomorphism*), which was one of the reasons such user interfaces became ubiquitous. For example, files are commonly represented as documents inside folders, much as they would be organized on an office desk. Following a similar analogy, to move files around the storage medium, a user picks them up and drags them to their new location. Similarly, to delete a file, a user drags it into a trash can. A

Pacemaker Programmer



Touchscreen and Joystick Operated Robot

Insulin Pump

Figure 2.2: *Examples of dedicated terminal user interfaces.*

paradigm that models reality made such UIs intuitive and easy to explain to users that are new to computers.

Graphical user interfaces rely heavily on a pointing device and allow the user to visually *recognize* operations. For example, a user can navigate a folder structure simply by clicking on the respective icons. In contrast, older command line interfaces allowed executing more advanced commands, but they also relied on *recollection* instead of recognition, as users had to already know the list of available command line commands, as well as their various options and arguments.

There are also classes of devices with user interfaces that do not follow the WIMP paradigm, and an example are modern mobile platforms (e.g., smartphones and tablets), as well as dedicated embedded terminals (ATM machines, pacemaker programmers, etc.).

Mobile platforms. Compared to desktop systems, the way users interact with mobile platforms is significantly different. Such mobile devices have a variety of sensors (ambient light, motion sensors, etc.), as well as a touchscreen instead of a mouse and keyboard combination. As a result, such devices support a richer set of input gestures, such as pinching, swiping, as well as multi-finger input. Furthermore, such devices have no standard notion of a "desktop", and commonly only have a single application running in the foreground at any given time.

Dedicated terminals. Contrary to such feature-rich devices and user interfaces, safety-critical systems (Figure 2.2) are commonly embedded terminals that run stripped-down operating systems. While on commodity devices a user can install and run various applications, and are designed to be applicable in a variety of settings and purposes, dedicated terminals are commonly designed only for one particular task (e.g., dispensing money,

programming implanted medical devices). The user interface design of such devices reflects those constraints, and the UI is comparatively simple, and does not consist of multiple windows the user can freely navigate. Such devices commonly execute only a single application, that does not provide the features of a modern desktop. Such constrains enable the introduction of more advanced security measures. Dedicated terminals can be configured to, e.g., prevent users from connecting their own (untrusted) input peripherals, or USB memory sticks. Critical terminals are often air-gapped, i.e., physically isolated from unsecured computer networks, to increase resilience against remote attacks. We defer a further description of smartphone user interfaces and dedicated terminals to Chapter 6 and Chapter 4, respectively.

The goal of a good user interface is to increase user productivity. That goal is commonly achieved through abstraction and simplification, as indeed, during daily interaction a user does not observe the full details of the inner workings of a computer system, due to underlying system complexity. Therefore, we can consider the user interface as a *window* into the system, where only the most relevant information for effective system control are presented to the user, and the remainder (e.g., complex program logic, internal variables) are hidden away.

2.2 Security of User Interfaces

User interfaces, by definition, are the only communication channels the user has with a computer system, based on which a user both decides future actions, as well as submits commands to the system. If those communication channels are maliciously tampered with, the overall security of a computer system can be compromised, irrespective of any applied architectural security features, e.g., system hardening [133, 143], process isolation, control-flow integrity [11, 60], or other security mechanisms [54, 71].

User interface attacks are possible in the absence of a secure, i.e., authenticated and confidential end-to-end channel, between the user and the application the user is interacting with. Lack of authentication enables attacks on input and output integrity (e.g, phishing attacks, command injection), and lack of confidentiality enables inference attacks (e.g., sensor-based approaches, shoulder surfing, etc.). Depending on the considered scenario, realizing such a channel can be challenging or even infeasible.

A potential solution to the problem is the concept of *trusted paths*. Based on the Trusted Computer System Evaluation Criteria [10] a trusted path is

"*a mechanism by which a person at a terminal can communicate directly with the Trusted Computing Base*".

A straw-man solution to achieve such a trusted path is to allow only signed and manually vetted applications to run on the system, as well as only use trusted peripherals. Such an approach can be feasible on dedicated devices, such as embedded terminals. However, on general purpose devices (e.g., smartphones) such an approach constrains the user and degrades user experience as users often install third-party applications. Simple forms of trusted path can also be achieved through *secure attention sequences*. For example, such a sequence on Windows OS is "*Ctrl+Alt+Del*", and it ensures that only the kernel can catch such combinations. However, such approaches implicitly trust input peripherals. Some approaches form the trusted path only within the borders of the system itself (e.g., they implicitly trust input and output peripherals [105]), while others extend the trusted path outside of the system by proposing custom input devices [128]. Some approaches work on mobile platforms [105], while others are designed for desktop systems [128, 189]. Furthermore, some are based on hypervisors [216] while others utilize hardware security features.

In terms of assumptions, trusted path approaches commonly enable secure input (and output) in the presence of a *malicious OS*. However, on their own, such approaches do not necessarily offer protection against other types of user interface attacks. For example, input inference attacks through side-channels may still be possible, depending on the considered system. In case the system does not rely on trusted peripherals, the attacker can still, e.g., replace the keyboard or device screen with malicious ones.

There are multiple ways of classifying related work in the field of UI security. One way is based on various trust assumptions (e.g., is the OS trusted, are the peripherals trusted, etc.). However, the goal of this thesis is to better understand the attack surface, as well as propose novel countermeasures, for *widespread* UI attacks, and we consider attacks that compromise the OS or introduce malicious hardware components (e.g., CPU, PCI, etc.) out of scope.

As various UI attacks were proposed that maliciously modify, inject, or infer either the commands a user is inputing into the system or the system's output that is presented to the user, we perform the classification from an attack-oriented perspective. Such attacks differ in attack goals, but commonly have a serious impact on users, as well as computer systems they target. In the remainder of this chapter, we review the various attacks on user interfaces, as well as their proposed countermeasures. We begin

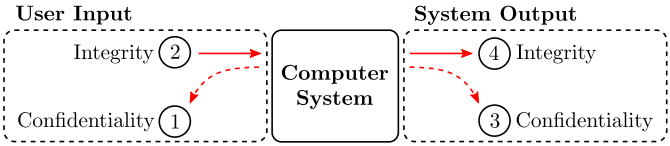


Figure 2.3: *High-level user interface attack goals. The attacker can compromise the confidentiality and integrity, of input or output, as well as combinations thereof.*

by stating the attacker model common to such attacks, and by creating a foundation on which to classify related work.

2.2.1 Adversarial Model

We observe from the user interface model presented in Figure 2.1 that the adversary has the following four high-level attack goals (Figure 2.3). The adversary can attempt to compromise the *confidentiality or integrity of user input*, as well as compromise the *confidentiality or integrity of system output*.

User interface attacks commonly assume the following model. The adversary either (1) controls a malicious application running on the target system, or (2) has physical access to the system. The former case commonly involves a malicious application that can either have some, or no special privileges. A common assumption is that the OS is not compromised, as a compromised OS can trivially inspect and modify all user input and system output. In the latter case, the adversary can approach the target system and, e.g., operate the device directly. The adversary can also attach a malicious hardware device to the target system, or in between an existing input peripheral (man-in-the-middle attacks). With this classification and adversarial model in mind, we proceed by summarizing related work.

2.3 Input Confidentiality

A class of user interface attacks, known as *input inference attacks*, focus on compromising the confidentiality of various types of user input, ranging from single screen taps, individual keystrokes, to typed words, whole sentences, or even spoken phrases. Such attacks are commonly performed by a malicious application running on the device that has access to device sensors. They can also be performed through dedicated hardware devices (e.g., hardware keyloggers) or other physical attacks (e.g., an attacker taking a picture of a locked device). Some attacks require special privileges

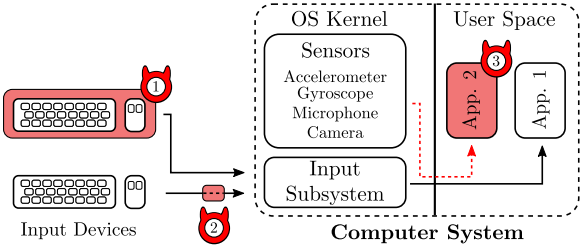


Figure 2.4: *Examples of user input attacks. Approaches based on malicious peripherals (1), man-in-the-middle devices (2), and various approaches where the malware is an application running on the device, e.g., sensor side-channels (3), were proposed. All three approaches can be used to compromise the input confidentiality, while approaches (1) and (2) can also be used to compromise input integrity, as described later in this chapter.*

(e.g., access to the microphone or camera), while some operate without any privileges at all.

2.3.1 Software Approaches

Microphone and camera approaches. Narain et al. [137] and Chowdhurry [44] propose attacks that infer both keystrokes and general screen tap locations using one or more smartphone microphones. Schlegel et al. [161] propose Soundcomber, where the authors access the microphone and perform light-weight speech recognition to extract sensitive information (e.g., credit card numbers), while the user is operating an interactive voice response systems. Simon and Anderson [167] demonstrate inferring PINs through the combined use of a front-facing camera, and a microphone, while Liu et al. [110] show how the technique of time difference of arrival (TDoA), commonly applied to ranging problems, can be used to infer typed keystrokes on a nearby, physical keyboard.

Fiebig and Hänsch [68] describe attacks where a malicious application extracts private information through the use of the front-facing camera. The camera is used to infer user keyboard input by observing facial reflections (e.g., from glasses on the user's face), as well as to steal user fingerprints when a user's fingertip moves through the camera's view.

Zhang et al. [209] take an image from an unprotected and inactive (e.g., locked) device. The authors then use computer vision approaches to extract fingerprints, and from the positions of those fingerprints the likely

unlock passcode. Yue et al. [200] take images while the user is typing (e.g., a PIN). The authors are unable to directly deduce typed input from the pictures, but can use computer vision techniques to estimate at which parts of the screen the input was performed. A possible defense for approaches based on the camera is to augment the smartphone with physical camera lids that would prevent unwanted screen grabs [68].

As the above approaches use sensitive peripherals, the malware requires special privileges in order to launch the attacks. Another set of approaches focus on motion sensors, and therefore do not require any special privileges, as access to those sensors is commonly allowed to all applications.

Motion sensor approaches. Cai and Chen [36] were the first to demonstrate the motion side channel on smartphones with on-screen numeric keyboards, and Aviv et al. [23] demonstrate using accelerometer data to infer user input. The authors focus on pin entry (numerical keyboards) and pattern entry scenarios. Wang et al. [186], and Sarkisyan et al. [159] demonstrate the same motion sensor side-channel on smartwatches.

Miluzzo et al. [134] and Xu et al. [196] use the combined readings of both accelerometer and gyroscope data to infer general smartphone screen tap positions. Owusu et al. [142] advance the attack, and demonstrate that accelerometer readings and a trained Random Forest model are sufficient to infer complete sequences of text, entered using the on-screen keyboard. Ping et al. [146] show how using motion sensors is sufficient to infer even longer user inputs (e.g., whole sentences).

As previous approaches commonly evaluate their attacks on a small number of devices, Cai and Chen [37] researched the practicality of such input inference attacks in a variety of settings, such as on multiple users, keyboard layouts, and smartphone models.

The applicability of such attacks is not limited to the device the malware is running on. Liu et al. [112] demonstrate how motion sensors in smartphones can be used to infer keystrokes on nearby physical keyboards. Marquaradt et al. [122] demonstrate that using smartphone accelerometer readings can be used to infer words typed on a nearby (physical) keyboard. Michalevsky et al. [132] showed that MEMS gyroscopes, embedded in modern-day smartphones, are sensitive to acoustic signals and can be used as a simple form of microphones. Using machine learning, the authors achieved 65% accuracy on detecting short voice phrases spoken near the phone (e.g., digit pronunciations). Input inference attacks were proposed that use sound to infer keystrokes inserted using physical keyboards [22, 217]. However, applying such approaches on smartphones is

challenging, due to the on-screen keyboard having different mechanical and acoustic properties [36].

The above inference attacks are presented in the context of Android, but such attacks work on other platforms as well. For example, Damopoulos et al. [51] demonstrate using motion sensors to infer taps on iOS platforms as well. Spreitzer [172] demonstrates that even data acquired from a low resolution sensor, i.e., the ambient light sensors of a smartphone, can be used to infer PINs typed on the same device.

As sensor side-channels are a common attack vector, a proposed defense is to reduce the amount of available information in the side-channel used in the attack by, e.g., reducing the sampling frequency of motion sensors [122, 132, 137, 142]. For example, an accelerometer could still be useful if it samples at 20Hz, instead of the common 100Hz, and reducing the sampling rate reduces the amount of useful information and would negatively affect inference precision. Fine-grained sensor access [142,161,196] (e.g., sensors require more elaborate permissions) is another possible defense. A more active defense approach is to randomly initiate the phone vibrator, to incur noise in the motion sensor data [142,166], or to frequently change sensitive input (e.g., create new passwords) [196], or even to prevent any access to motion sensors while the user is performing a critical operation (e.g., keyboard input) [23, 134]. Another defense approach is to randomize the keyboard layout during sensitive input operations (e.g., passwords) [169], but such approaches can incur user errors, and can negatively affect user experience.

Elevate privileges. A malicious application could also attempt to elevate its privileges through vulnerabilities present in higher-privileged applications, or the operating systems itself and thereby gain direct access to user input. However, such software attacks are outside of the scope of this thesis.

Other approaches Diao et al. [57] take a different approach, and propose a novel side-channel based on interrupt timing analysis. The authors observe that the interrupt number patterns (the amount of interrupts per unit of time) leaks information about the input. Such information is available through the `proc` filesystem on Android, and the authors use the data to infer unlock patterns as well as which application is running in the foreground. Authors propose decreasing the resolution of interrupt data, as well as finer-grained access control to the `proc` filesystem [57] as possible countermeasures.

2.3.2 Physical Approaches

A physically proximate attacker can attempt shoulder-surfing attacks, where an attacker attempts to compromise the confidentiality of user input by simple observing the user typing the input. Such attacks can be prevent by shoulder-surfing safe login methods [145].

An attacker can also compromise the confidentiality of user input through hardware attacks, such as through the use of a hardware keylogger [98] that records all inserted keyboard input. Other than keyloggers, an attacker can also use more complex man-in-the-middle hardware [99]. However, as such types of attack are more closely related to compromising integrity of input, we summarize them in the upcoming section. To prevent hardware-based attacks, one can use dedicated (trusted) devices [127], or software attestation of peripherals [106]. In such cases, the peripherals can share keys with the host, and ensure confidentiality and authenticity of input data between the peripheral and host system.

For example, Bumpy [128] is a system for creating a secure input path between user input peripherals, and a remote web service. The system proposes the usage of specialized hardware that encrypts all user input prior to sending it through USB. Inside the OS, trusted code running inside the Flicker [126] framework decrypts the input, and releases it to the destined application. Through the use of a dedicated device, the system also informs the user to which website (if any) the input is going to. However, the Bumpy system was designed for desktop scenarios, and not mobile smartphone platforms.

Summary. Input inference approaches based on motion sensors, microphones, or cameras, suffer from noise in the channels and are inherently sensitive to environmental factors, e.g., if the user is moving, or navigating a noisy environment. For example, microphone based keystroke inference [137] works well only when the user is typing in portrait mode. As a result, they either suffer from accuracy issues and require multiple observations to boost the inference accuracy, or require additional permissions that could raise user suspicion.

2.4 Input Integrity

In the previous section we discussed approaches that focus on compromising confidentiality, and we can consider such approaches as *passive*, as they only attempt to infer user input. In this section we discuss *active* attacks that compromise the integrity of user input by injecting malicious

commands into the target system.

Directly operate the user interface. A simple attack vector that compromises the integrity of user input is when an attacker has physical access. The attacker approaches the device, and directly operates the user interface of the target system (e.g., by using the available keyboard or mouse). The attacker can then perform any action that the user interface exposes. Instead of only operating the UI, an attacker can also attempt to install malware onto the system itself by, e.g., accessing a command console, or uploading the malware through USB peripherals. However, such manual attacks take time to execute, and may not always be applicable (e.g., dedicated terminals can prohibit executing untrusted applications). Furthermore, such attacks can be prevented through the use of authentication (e.g., passwords, second-factor dongles, etc.).

The Universal Serial Bus (USB) implies trust between host and device. While the recent USB 3.0 defined an authentication protocol, prior version of the standard does not require (or provide mechanisms) to authenticate or attest peripherals or the host. USB implementers forum considers users to be responsible for their own security [113], stating that "*[...] consumers should always ensure their devices are from a trusted source and that only trusted sources interact with their devices.*". A feature of USB is the support for composite devices. Such composite devices enable a single device to simultaneously expose the interfaces of multiple devices (e.g., a microphone and a webcam). For example, a malicious USB memory stick can announce itself to the target system as a keyboard. As a result, physical attacks were proposed where an attacker attaches a malicious device to the system, or compromises an already attached one.

Command injection. In command injection attacks, a malicious attack device commonly masquerades as a human interface device (HID), such as an USB mouse or keyboard [26, 77, 84, 140]. The attack device then injects malicious input that, e.g., adds a new administrative user, or executes an application and infects the target system with malware. An example are BadUSB [140] attacks, where the adversary compromises the firmware of a legitimate USB peripheral (e.g., memory stick), and augments it with malicious logic. Instead of compromising existing peripherals, the attacker can also use a custom device (e.g., rubber ducky [84], facedancer [77]), dedicated to performing command injection attacks. Kierznowski proposes BadUSB 2.0 [99], a more advanced version of the attack where the attack

device is not only connected to the target system, but is located in between the legitimate peripheral (man-in-the-middle).

Command injection attacks are a realistic threat, as studies showed that many people (as high as 98%) plug in unknown USB devices found on the street [94, 182]. Although such attacks target desktop systems, the approach of injecting commands via malicious peripherals conceptually applies to any other system with exposed peripheral ports. For example, Wang and Stavrou [187] demonstrate how a phone can be compromised by injecting input commands through the connected USB cable.

2.4.1 Proposed Defenses

A proposed defense against command injection attacks on platforms that contain motion sensors (e.g., smartphones) is through user identification, e.g., by using motion sensors to identify users based on tap patterns [210], or movement data [33]. However, a malicious device could attempt to learn the input patterns of a real user prior to launching the attack. Another possible defense is through the use of device authentication where, e.g., the host only accepts a trusted and limited set of devices. A simple form of device authentication is through the use of vendor or product IDs. However, those identifiers are trivial to spoof by a malicious device. Another is to white-list devices based on serial numbers, however, firmware can be compromised and report spoofed serial numbers.

User-centric defenses. There are various user-centric defense approaches that prompt the user to detect malicious behavior. In USB Keyboard Guard [52] for example, the authors propose capturing every event when a new keyboard device is added. The system then prompts the user with a screen, and instructs the user to decide whether or not to add the keyboard device. In GoodUSB [179], Tian et al. propose a software-based approach for isolating USB devices from the host. The system requires kernel modifications, and it prompts the user on every new device connection. Potentially malicious (denied) devices are redirected to an internal honeypot for further monitoring.

Kang [96] proposes USBWall, a hardware based approach where a physical separation between host and peripheral is introduced by a specialized device. Any USB device is considered as untrusted, until it specifies its capabilities (intentions), and then the user decides wether to add the device or not. This approach is conceptually similar to SandUSB [114]. Griscioli et al. [82] propose a hardware device that forces the user to interact with

any newly connected USB devices, prior to their usage (authorization). For example, if a new keyboard is attached, the user is required to type a random challenge through the device. Similarly for a connected mouse, the user is required to draw a line between two selected points on the screen.

The drawback of *user-centric* approaches is that users can get desensitized to the security prompts over time, and begin to habitually accept arbitrary devices. A similar problem is known in the context of security indicators, as research has shown that even when users were instructed to pay attention to security indicators, many failed to do so, while some disregarded them completely [192], due to not understanding the security implications of their actions. To remove the user from the decision process, *automated* approaches (e.g., based on access control) were proposed.

Access control approaches. Tian et al. [180] propose USBFilter, a firewall system for USB devices. The authors presented a tool that enables pinning file descriptors to certain processes. For example, access to a specific USB device (e.g., USB microphone), which is a file on Linux, can be restricted to a particular process (e.g.. VOIP application). Similar pining of files to specific applications can also be performed by alternative approaches such as PinUp [59] and SELinux [148]. Furthermore, by performing isolation on a low level, USBFilter can protect (possibly flawed) kernel drivers from malicious peripherals, as it is blocked before the required drivers are loaded. The approach also enables the ability to filter based on specific interfaces of a single device — an approach relevant for malicious composite devices.

Angel et al. [19] propose Cinch, a virtualization layer to secure USB communication by separating an untrusted machine from a protected one, through the use of a hypervisor. The authors also propose a overlay protocol, that enables establishing TLS sessions between the USB device and the target system. In such a manner, access can be denied to untrusted devices (devices that don't possess the signed certificate). However, implementing such a cryptographic overlay requires modifying USB devices.

Wang and Stavrou [188] propose USBSec, a custom authentication layer on top of USB. However, such an approach requires custom kernel extensions. Furthermore, there are various open-source [100], as well as commercial projects [47, 48, 74] that offer access control functionalities for USB devices.

Other approaches. Bates et al. [28] and Letaw et al. [104] propose using USB ports to determine identity through the use of fingerprinting. The authors show that a USB device can identify the host, based on features

inherent to the machine hardware, or USB software stacks. In such cases, a peripheral device (e.g., a mouse) can detect if a malicious man-in-the-middle device (e.g., BadUSB2.0 [99]) is present, or if the device is directly communicating with the host. Barbhuiya et al. [26] propose an approach based on analyzing keystroke dynamics, and detecting input that was created in an automated manner (e.g., by a malicious peripheral, as opposed to user-typed input). Another approach to protecting against malicious firmware modification attacks [140] is through cryptographically signing firmware images. However, such approaches can require complex key management. Additionally, to prevent command injection attacks one can use software attestation to, e.g., only allow the use of trusted input peripherals [106].

An alternative approach is to randomize keyboard keys, however such approaches are commonly applicable only to systems where the keyboard is an application running on the system (e.g, in case of smartphones). The rationale is that an automated input mechanisms (e.g., malicious peripheral posing as a mouse) does not know the current keyboard layout. However, such an approach only protects against keyboard input through the on-device keyboard, as the malicious peripheral could also register a separate USB keyboard device.

Summary. Currently proposed command injection attacks are limited to executing simple sets of pre-programmed user input [77, 84, 99, 140]. To detect such attacks, a common assumption is that malicious peripherals will exhibit some form of non-compliant behavior. For example, if a storage peripheral registers itself as an input peripheral to inject user input, or if the automated input is inserted at a different speed or style [26]. It is therefore challenging to apply existing approaches to detect attacks that are *compliant*, e.g., an attack that does not create an additional input channel (e.g., registers a new keyboard), but rather injects input through the regular input channel (e.g., in the case of a man-in-the-middle device [99]).

2.5 Output Integrity

In the previous sections we focused on user input attacks. However, an attacker can also directly attack the system output as well. A type of such attacks that compromise the output integrity are also known as *user deception attacks*. We can think of many attacks that compromise *integrity of output* (e.g., phishing) as attacks whose final goal is to compromise *integrity*

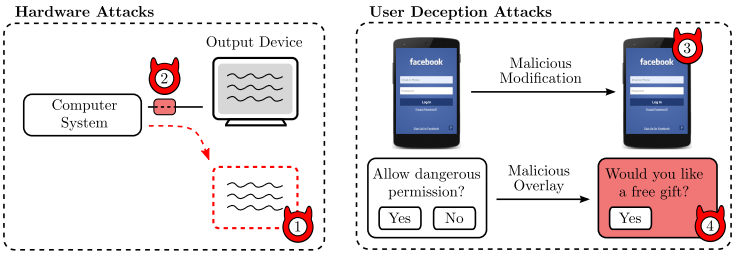


Figure 2.5: *Examples of system output attacks. Approaches that compromise the confidentiality of output based on side-channels such as RF leakage and shoulder-surfing attacks (1), or man-in-the-middle devices (2) are possible. Furthermore, attacks on output integrity, such as user deception attacks, e.g., spoofing (3) or UI redressing (4), were proposed.*

of input (described in the previous section). However, we decouple those two sections for clarity of presentation.

2.5.1 User Deception Attacks

Phishing attacks are an all too common occurrence, and are good examples how efficient and devastating user interface attacks can be. By maliciously changing what the user sees (e.g., the contents of the device screen), such user deception attacks attempt to confuse the users into believing that they are interacting with a known and trusted application, when in reality they are interacting with a another one. Such attacks assume an adversary that controls a malicious application on the target device.

An example of such attacks are spoofing attacks, where a modified stock market application changes the value of the current stock prices that the user sees. The application is otherwise completely benign, and by modifying what the user sees such an app then deceives the user into, e.g., selling or buying stock. A more common form of such spoofing attacks are *phishing attacks*, where a malicious application creates a fake interface, commonly for the purpose of stealing user credentials.

The existing application phishing detection systems attempt to identify API call sequences that enable specific phishing attacks vectors (e.g., activation from the background when the target application is started [31]). Many approaches in web phishing detection analyze the contents of the website's DOM tree, and compare the structure and elements to know instances of phishing attacks [12, 111, 154, 205, 208]. Such approaches could be also applied to smartphone scenarios, as mobile operating systems

commonly keep all UI elements in a tree structure similar to a HTML DOM tree. Another approach is to consider the visual presentation of a spoofing application (or a website), and compare its similarity to a reference value [43, 72, 125]. Approaches based on Gestalt theory [190] to detect web phishing using visual similarity were proposed [43], as well earth mover's distance [72] were proposed. There are various attack vectors that enable such phishing attacks to occur, and we summarize *repackaging*, as well as *activity* and *task hijacking*.

Application repackaging is a simple and widespread [76, 214] malware deployment vector, where the attacker downloads a legitimate application (commonly from a marketplace), appends malicious logic, and re-uploads it to the marketplace under a different author. Such modifications can easily be performed at large scale, e.g., by use of automated tools.

Activity hijacking is an attack vector where an attacker brings a new activity to the foreground [42, 63] (e.g., a malicious copy of the previously active app). Such attacks consist of two steps: (1) detect when the targeted app is active, and (2) place the malicious activity in the foreground. The first step can be realized through side-channels such as the `proc` filesystem [42], or from design features of the windowing manager [150]. An alternative form of the attack is through "immersive" fullscreen windows [31] that enable creation of fullscreen windows the user can not escape out of.

Task hijacking are recent attack vectors for user deception attacks. They utilize user interface design features (specific flags of the underlying GUI subsystem, the design of the activity stack, the way the "Back" button is implemented, etc.) to change the order of the window stack [150]. After the attack is launched, by clicking the "Back" button, the user brings a malicious activity to the foreground, instead of the expected (benign) one.

User interfaces on smartphone platforms are sophisticated subsystems. They offer a variety of APIs that an attacker can use to divert the normal flow of UI operations to, e.g., bring a malicious phishing window in the foreground without the user noticing the transition. Furthermore, such attacks are applicable to any UI on the system, including system applications. Although the above attacks were proposed in the context of Android, similar attacks are shown to work on iOS as well [156].

Other attacks. Tapjacking (or clickjacking) [90, 116, 139, 152, 156] are attacks where the user believes that one application is receiving their input,

when in reality, the input is processed by another app. For example, the user see and presses the Facebook "Like" button, when in reality, e.g., the user allowed the attacker administrative access to the device.

Other UI attacks are also possible. For example, if a malicious app shows a static image of a UI, the attacker can cause a denial-of-service effect on the user, as the user sees the UI but is unable to interact with it, and concludes that the app is unresponsive.

2.5.2 Proposed Defenses

User deception attacks are a major threat [83] and one reason why they are so successful, is that users can not determine the provenance of the content presented on the device screen. Felt and Wagner [63] pointed out that the lack of trusted path from input to the app is both a problem, as well as the reason for phishing attacks. Another reason is that upwards of 90% of users use the visual look and feel of a website as means of authenticity [56, 95]. The login screen the users see could have been created by a benign and trusted application, just as it could have been created by a malicious one. As a result, approaches that aid the user in detecting such attacks were proposed.

Existing repackaging detection approaches mostly use fingerprinting based on static (such as code structure [65] or imported packages [215]), and dynamic features [21, 75, 147, 184, 198] (such as system call [108] or network traffic patterns [163]). For example, EagleDroid [175] extracts fingerprints from Android application layout files, while ViewDroid [204] and MassVet [41] perform static analysis on parts of the code relevant to the UI construction to extract a graph that expresses the user interface state and transitions. The rationale behind these works is that while application's code can be easily obfuscated, the user interface structure must remain largely unaffected.

User-centric approaches. Approaches based on security indicators [31, 103, 121, 162] can help users to detect deception attacks. Such approaches either annotate parts of the display presented to the user, or use dedicated hardware devices, in order to inform the user when interaction with a trusted (or untrusted) application is taking place.

A possible approach is to permanently dedicate a part of the smartphone device screen to security indicators [63]. For example, Lange and Liebergeld [103] propose Crossover, an approach that dedicates a part of the screen to indicate which virtual mxachine (VM) is currently active on a mobile platform that supports multiple VMs. Similarly, Selhorst et al. [162]

propose “m-gui”, a secure GUI that always shows the name of the currently active application (compartment) at the top of the screen. The drawback of such approaches is that mobile devices are limited in terms of available display size. Furthermore, such approaches require attentive users to notice the indicators.

Bianchi et al. [31], and similarly Fernandes et al. [67], propose a security indicator located inside the bottom navigation bar (next to the “Back” key). The indicator always presents which application is currently active. However, such an approach is susceptible to attacks where an attacker draws over the security indicator (e.g., in full-screen apps). For that purpose, the authors also propose using a secret image, that the attacker does not know and therefore cannot spoof. Bumpy [128] uses a small dedicated screen as a form of security indicator. Fernandes et al. [66] propose AuthAuth, a trusted input path where the authors annotate the system keyboard app with secret images, known to the user, but not to an attacker attempting to launch spoofing attacks. Based on such images, the user can visually check if the input is going to the intended application. GuarDroid [183] takes a similar approach of annotating the keyboard with secure phrases. Furthermore, GuarDroid establishes a trusted path for password entry between potentially untrusted apps running on the smartphone, and an external service. The OS encrypts all user input (e.g., password typed through the on-device keyboard) prior to sending it to the corresponding user-land application. A legitimate app would then send this input through the network to a remote service, e.g., to login. The OS would intercept the network transmission, and replace the encrypted password with the correct one. A malicious phishing application therefore only sees the encrypted text input, and not the real password.

The drawback of approaches based on security indicators, is that they rely on users to be constantly attentive and able to detect signs of attacks taking place (e.g., visual discrepancies in the security indicators).

UI hardening approaches. Chen et al. [39] focus on analyzing and preventing sensitive input leakage from input method editor (IME) apps, such as custom keyboard apps. Zhang et al. [206] propose App Guardian, a system that automatically pauses suspicious background apps while sensitive input is taking place. Ren et al. [149] propose WindowGuard, a system capable of preventing a wide range of UI redressing attacks. Roesner and Kohno [152] consider the problem of securely embedding third-party user interface components (e.g., ad libraries) into existing apps. The authors propose LayerCake, a modified Android framework that prevents one prin-

cipal (UI element) from maliciously tampering with other elements.

Summary. Spoofing attacks, such as phishing, remain a widespread and devastatingly efficient user interface attack. Existing prevention mechanisms based on security indicators suffer from lack of user attention.

2.6 Output Confidentiality

An attacker can compromise the confidentiality of system output (Figure 2.5) in various ways. An attacker can compromise the link between the system and the output device (e.g., a monitor), or simply observe the system output through shoulder-surfing attacks. However, such attacks can be addressed using special head-mounted displays and appropriate coding [109, 193]. Another approach is to exploit vulnerabilities in the monitor's firmware, attach a malicious man-in-the-middle device, or use side channels to infer the presented data. One such side channel is based on electromagnetic emanations [102], where the attacker reconstructs the video image from the signals that leak through the cable connecting the graphics card to the monitor. Such attacks can be prevented by shielding the cables [7] or by fortifying the security of the monitor's firmware [11, 54, 60, 71, 133, 143]. An alternative physical attack is to intercept hardware components en-route [9]. An attacker could then replace legitimate components with their malicious counterparts.

Such attacks are realistic and potentially damaging, but are also highly specialized, and fall outside of the scope of this thesis. We therefore do not focus on them any further.

2.7 Summary

The goal of this thesis is to both advance the state-of-the-art in user interface attacks, as well as to propose novel defense mechanisms against widespread types of user interface attacks. Towards that end, in this chapter we motivated the problem of user interface security, we reviewed related work and pointed out the benefits and drawbacks of existing attacks as well as defense approaches.

We conclude this chapter with a visual representation of key parts of the UI security problem space (Figure 2.6). The goal of the figure is to visually position the contributions of this thesis with respect to broad related work areas, and also to serve as a roadmap for upcoming chapters.

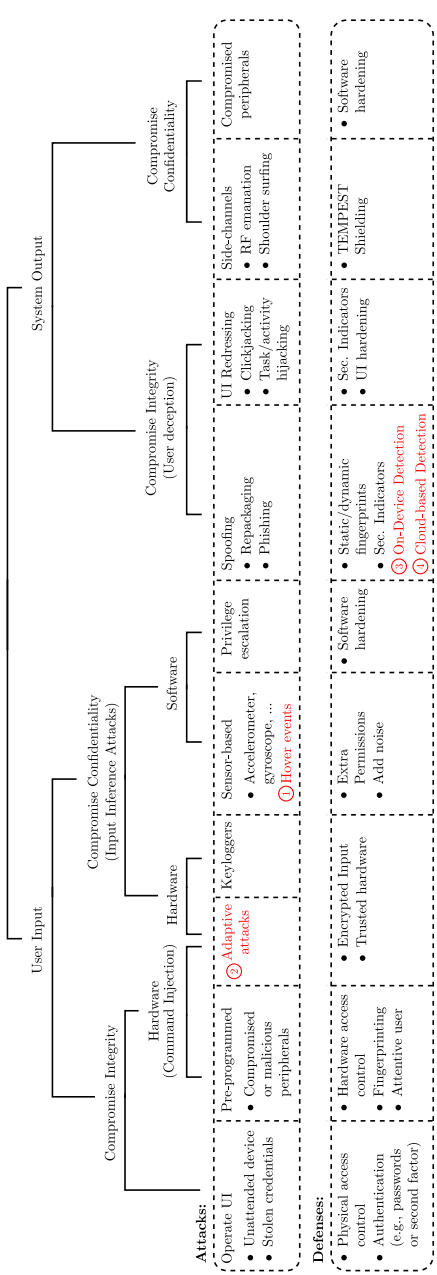


Figure 2.6: *Thesis roadmap and high level overview of key areas of related work. The contribution of this thesis (marked in red, and numbered based on the chapter they appear in) are two novel user interface attacks (hover and adaptive attacks), and two novel spoofing detection systems for smartphones.*

Part I

Attacks

Introduction

In Chapter 2, we reviewed related work in the domain of user interface attacks, and we saw that various input inference attacks were proposed. Some used motion sensors, while others used microphones or cameras. However, such attacks suffer from sensitivity to noise in the employed side-channels (or environment), or only work in specific scenarios (e.g., when the device is in portrait mode). Furthermore, existing approaches either suffer from accuracy issues, or require additional permissions that could raise user suspicion.

One reason why side-channels, the common sources of such attacks, are challenging to prevent is that they are difficult to account for at system design time. System architects nowadays can model and prevent well-known side-channels, such as timing [97], temperature [124], RF [102] and acoustic [25] emanations. Furthermore, the existence of side-channels is highly specific to the computer system at hand, as some side-channels that exist on smartphones (e.g., using ambient light sensors to infer PINs) may not necessarily exist on other devices as well.

In Chapter 3, we present a novel Android input inference attack that is based on a new side-channel (hover). Our attack overcomes the limitations of prior work, and enables precise and continuous input inference.

Furthermore, we observed that existing command injection attacks are limited to only executing simple sets of pre-programmed user input (e.g., open command console, add new administrative user). In Chapter 4 we therefore present a novel class of adaptive command injection attacks that target both dedicated terminals, as well as general purpose devices (e.g., desktop systems). Our attacks enable new attack scenarios that are beyond the ones of simple command injection, and to which existing command injection attacks attacks are not applicable to.

Chapter 3

Inferring User Input with Hover

3.1 Introduction

The recent years witnessed a surge of user input inference attacks. This is not surprising, as these attacks can profile users or obtain sensitive user information such as login credentials, credit card numbers, personal correspondence, etc. Existing attacks are predominantly application-specific, and work by tricking the users into entering their information through phishing or UI redressing [31, 139, 150, 152, 191]. Other attacks exploit readily available smartphone sensors as side-channels. They infer user input based on readings of various sensors, such as the accelerometer [85], gyroscope [132] and microphone [137]. Access to these sensors (microphone excluded) requires no special permissions on Android.

In this chapter, we introduce a novel user input inference attack for Android devices that is *more accurate*, and more general than prior works. Our attack simultaneously affects all applications running on the device (it is *system-wide*), and is not tailored for any given app. It enables continuous, precise collection of user input at a high granularity and is not sensitive to environmental conditions. The aforementioned approaches either focus on a particular input type (e.g., numerical keyboards), are application-specific, operate at a coarser granularity, and often only work under specific conditions (limited phone mobility, specific phone placement, limited environmental noise). Our attack is not based on a software vul-

nerability or system misconfiguration, but rather on a new and unexpected use of the emerging *hover* (floating touch) technology.

The hover technology gained popularity when Samsung, one of the most prominent players in the mobile market, adopted it in its Galaxy S4, S5, and Note series. The attack presented in this chapter can therefore potentially affect millions of users [29, 30, 70, 92]. The hover technology produces a special type of event (hover events) that allow the user to interact with the device without physically touching its screen. We show how such hover events can be used to perform powerful, system-wide input inference attacks.

Our attack carefully creates and destroys overlay windows, right after each user's tap to the foreground app, in order to capture just enough post-tap hover events to accurately infer the precise click coordinate on the screen. Previous phishing, clickjacking, and UI redressing techniques [31, 139, 150, 152, 191] also create such overlay windows, commonly using the `SYSTEM_ALERT_WINDOW` permission. Our attack benefits from, but does not require such a permissions, as we present an implementation that does not require *any permissions*. Furthermore, overlay windows in our case are exploited in a conceptually different manner; our attack is continuous, completely transparent to the user, does not obstruct the user's interaction with the foreground app, does not redirect the user to other malicious views, and does not deceive the user in any manner — a set of properties not offered by existing attacks.

To evaluate our attack, we implemented *Hoover*, a proof-of-concept malicious application that continuously runs in the background and records the hover input of all applications. However, to realize our attack we had to overcome technical challenges. Our initial experiments with the hover technology showed that hover events, unexpectedly, are predominantly not acquired directly over the point where the user clicked. Instead, the events were scattered over a wider area of the screen. Therefore, to successfully predict input event coordinates, we first needed to understand how users interact with smartphones. For this purpose, we performed a user study with 20 participants interacting with one of two devices with *Hoover* on it, in two different use-case scenarios: general clicking on the screen and typing regular English text. The hover events acquired by *Hoover* were then used to train a regression model to predict click coordinates, and a classifier to infer the keyboard keys typed.

We show that our attack works well in practice with both stylus and fingers as input devices. It infers general user finger taps with an error of 100px. In case of stylus as input device, the error is reduced to just 2px.

Whereas, when applying the same adversary to the on-screen keyboard typing use-case, the accuracy of keyboard key inference results of 98% and 79% for stylus and finger, respectively.

A direct and intuitive implication of our attack is compromising the confidentiality of all user input, system-wide. For example, *Hoover* can record various kinds of sensitive input, such as pins or passwords, as well as social interactions of the user (e.g., messaging apps, emails). However, there are also alternative, more subtle, implications. For example, Hoover could potentially profile the way the device owner interacts with the device, i.e., generate a biometric profile of the user. This profile could be used to, e.g., restrict the access only to the device owner, or to help an adversary bypass existing keystroke based biometric authentication mechanisms.

We discuss possible countermeasures against our attack, and we observe that, what might seem as straightforward fixes, either cannot protect against the attack, or severely impacts system or hover technology usability.

3.2 Background on Android

In this section we provide a brief primer on the hover technology and on alert windows, a common UI element used by many Android mobile apps.

Hover events in Android. The hover (or floating touch) technology enables users to interact with mobile devices without physically touching the screen, and we illustrate the concept in Figure 3.1. The technology was first introduced by the Sony Xperia smartphone in 2012 [170], and is achieved by combining mutual capacitance with self-capacitance sensing systems. After the introduction by Sony, the hover technology was adopted by Asus in its Fonepad Note 6 device in late November 2013. It finally took off when Samsung, one of the biggest players in the smartphone market, adopted it in a series of devices including the Galaxy S4, S5, and the Galaxy Note [158]. Samsung alone has sold more than 100 million devices supporting the hover technology [29, 30, 70, 92], and all these devices are a potential target of the attack described in this chapter.

The hover is handled by the system as follows. When the user interacts with the smartphone, the system can detect the position of the input device before it touches the screen. In particular, when the input device is hovering within 2cm from the screen (see Figure 3.1), the operating system triggers a special type of user input event (the hover event) at regular intervals. Apps that catch the event gain from it the location of the input device over the screen in terms of x and y coordinates. Once the position of the input

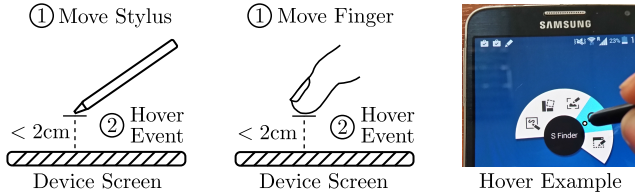


Figure 3.1: *Hover (floating touch) technology. The input device creates a special class of events (hover events) without physically touching the device screen. The rightmost part shows the hover technology in use. The user is interacting with the phone without the input device physically touching the device screen.*

device is captured by the screen's sensing system, it is then dispatched to the `View Objects`, Android's building blocks for user interface, listening to the event. In more details, the flow of events generated by the operating system while the user hovers and taps on the screen are as follows. The system starts firing a sequence of *hover events* with the corresponding (x, y) coordinates, when the input device gets close to the screen (less than 2cm). A *hover exit* event followed directly by a *touch down* event is fired when the user touches or taps on the screen, followed by a *touch up* event notifying about the end of the touch. Afterwards, another series of *hover* events with relative coordinates are again fired as the user moves the input device away from the original point of touch. Finally, when the input device leaves the hovering area, i.e., is floating higher than 2cm from the screen, a *hover exit* event is fired.

View Objects and Alert Windows. The Android OS handles the visualization of system and application UI components on screen through the `WindowManager` interface [15]. This interface is responsible for managing and generating the windows, views, buttons, images, and other floating objects on the screen. Depending on their purpose, the views can be generated as to catch hover and touch events (active views, e.g., a button), or not (passive views, e.g., a static image). However, the mode of a given view can be changed, e.g., from passive to active, through specific flags of the `updateViewLayout()` API of the `WindowManager` interface. For example, to make a view passive we can set the `FLAG_NOT_FOCUSABLE` and `FLAG_NOT_TOUCHABLE` flags. The first flag avoids that the view blocks possible touch events destined for textboxes of other apps that are un-

derneath the view. The second flag disables the ability to intercept any touch or hover event. Setting the two flags prevents the static view from interfering with the normal usage of the device, even though it is always present on top of any other window. In addition, a given view can know when a click was issued somewhere on screen and outside the view area, i.e., without knowing the click position. This is made possible by setting the `FLAG_WATCH_OUTSIDE_TOUCH` parameter of the view.

Alert windows are views that, even when created from a background service, the OS puts on top of every other visible object, including those of the app the user is currently interacting with [16]. To generate alert windows, the `WindowManager` interface utilizes the `SYSTEM_ALERT_WINDOW` permission, that the service that creates the view must hold. This permission is used by system apps such as "Text Messaging" or "Phone" to show information related to the cost of the last text message or telephone call. Most importantly, the permission is very common to many apps on Google Play, as it enables users to quickly interact with a given app while they are using another one. An example is the Facebook Messenger's "chat head" feature, that let the user reply outside the Messenger app. Among the popular apps that use the `SYSTEM_ALERT_WINDOW` permission are also Facebook, Skype, Telegram Messenger, etc. These apps alone have recorded billions of installs on Google Play.

3.3 Our Attack

The goal of our attack is to track every click the user makes with both high precision (e.g., low estimation error) and high granularity (e.g., at the level of pressed keyboard keys). The attack should work with either finger or stylus as input device, while the user is interacting with a device that supports the hover feature. Furthermore, the attack should not be detected by the user, i.e., the attack should not obstruct normal user interaction with the device in any way. Before describing our attack, we state our assumptions and adversarial model.

3.3.1 Assumptions and Adversarial Model

We assume the user is operating a mobile device that supports the hover technology. The user can interact with the mobile with either a stylus, or a single finger, without any restrictions.

We consider the scenario where the attacker controls a malicious application installed on the users device, and where the goal is to violate the confidentiality of user input without being detected. The malware can have

access to the SYSTEM_ALERT_WINDOW, but does not require it. Furthermore, we assume the malware has the INTERNET permission that Android designated as a PROTECTION_NORMAL permission [18] that is granted to all apps that require it at install time.

3.3.2 Attack Overview

To track the input device immediately after a user performs a click, we exploit the way Android OS delivers hover events to apps. When a user clicks on the screen, the following sequence of hover events with corresponding coordinates and time stamps are generated (see Section 3.2): hover events (input device floating), hover exit and touch down (on click), touch up (end of click), hover events (input device floating again).

The above sequence already shows that just by observing hover events one could infer when and where the user clicked. To obtain these events, a malicious application can generate a transparent alert window overlay if it holds the SYSTEM_ALERT_WINDOW permission, otherwise it can, e.g., use the Toast class to create the overlay and implement the attack as described in Section 3.4.5. Recall that the alert window components are placed on top of every other view by the Android system (Section 3.2). Once created, the overlay could catch the sequence of hover events fired during clicks and would be able to track the input device. However, doing so in a stealthy way without obstructing the normal app interaction of the user (the foreground victim application) is not trivial. The reason is that Android sends *hover events* only to those views that receive *touch events*. In addition, the system limits the consumption of a touch stream of all events in between and including *touch down* and *touch up*, to one view only. Therefore, a transparent and malicious overlay tracking the input device would either catch both hover events and the touch, thus preventing the touch to go to the victim app, or none of them, thus preventing the malware to infer the user input.

3.3.3 Achieving Stealthiness

The malicious app controlled by the adversary cannot directly and stealthily observe click events. We show that it can instead infer the clicks stealthily by observing hover events preceding and succeeding each user click. In doing so, the adversary is able to infer the user input to the device without interfering with regular user interaction, and our attack works as follows.

The malicious app generates a fully transparent alert window overlay which covers the entire screen. The overlay is placed by the system on top

of any other window view, including that of the app that the user is using, and due to the overlay the malware can track hover events. However, the malicious view switches from *active* (catch all events) to *passive* (pass them through to the underneath app) in a smart way, so that the touch events go to the real app while the hovering coordinates are caught by the malware. The malware achieves this by creating and removing the malicious overlay in a way that does not interfere with the user interaction. We proceed by describing the procedure in the next section.

3.3.4 Catching Click Times and Hover Events

We implement our adversary (malware) as a background service that is always running on the victim device. The main challenge of the malware is to know the exact time when to switch the overlay from *active* (add it on screen) to *passive* (remove it), and back to *active* mode again. Note that, to guarantee the attack stealthiness, we can only catch hover events, as catching other types of events (e.g., user taps) would result in disrupting the user's regular UI experience. Therefore, estimating when the user is going to stop hovering the input device in order to perform the click on the screen is not trivial, and we approach the issue in the following way.

The malware makes use of two views through the `WindowManager`. One is the fully transparent alert window overlay mentioned earlier in this section. The second view, that we call the *listener*, has a size of 0px and it does not catch hover events nor regular user clicks. Its only purpose is to give the malware knowledge of when a click happens, and the Hoover malware will then use this information to remove (or recreate) the transparent overlay accordingly. The listener view has the `FLAG_WATCH_OUTSIDE_TOUCH` flag activated, which enables it to be notified each time a click happens anywhere on the screen. Then, the malware engages the two views during the attack as follows.

Inferring click times. Every user click happens outside the listener view, as the view has a size of 0px. In addition, this view has the appropriate flag `FLAG_WATCH_OUTSIDE_TOUCH` set, and is therefore notified by the system when the click's corresponding *touch down* event is fired. As a result, the malware infers the exact timestamp of the click, though it cannot know its position on the screen just yet (step 1 in Figure 3.2).

Catching post-click hover events. After the click time is detected, the next step is to infer the click position. This is done through the activation of the whole-screen transparent overlay, which is then used to acquire

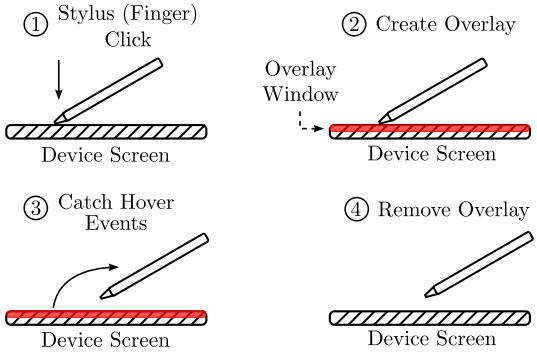


Figure 3.2: *Post-click hover events. Hoover catches post-click hover events through the use of a transparent malicious overlay.*

the hover events that succeed the click. However, the malware activates the overlay only after the previous phase; namely after the *touch down* event has already been fired and the click has already been intercepted by the application the user was interacting with (step 2 in Figure 3.2). This guarantees that the attack does not interfere with the normal user experience. From that moment on, the overlay intercepts hover events fired as the input device moves away from the click's position and towards the next screen position the user intends to click on (step 3 in Figure 3.2).

In contrast with the listener view, which cannot interfere with the user-device interaction but can be constantly present, the overlay cannot be present on screen, otherwise it will obstruct the next clicks the user makes. At the same time, the overlay must remain active long enough to capture a sufficient number of hover events succeeding the click to perform an accurate click location inference. With the devices considered in this work, our experiments show that hover events are fired by the operating system once every 19ms on average. In addition, we find that an activation time of 70ms is a good trade-off between catching enough hover events for click inference and not interfering with the user-device interaction. After the activation time elapses, the malware removes the overlay again (step 4 in Figure 3.2).

3.3.5 Inferring Click Positions

At this stage, the malware has collected a set of post-click hover events for each user click. Starting from the information collected, the goal of the attacker is to infer the position of each user click as accurately as possible.

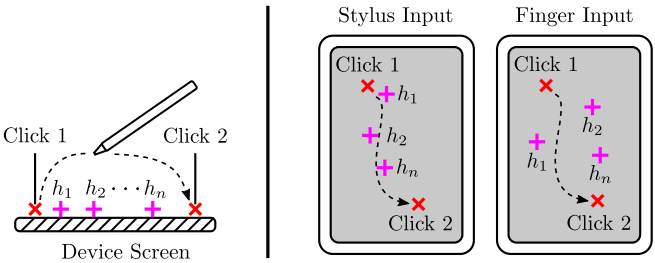


Figure 3.3: *Example of post-click hover events collected by Hoover. In case of stylus input, the captured post-click hover events ($h_1, h_2, \ldots, h_n$) tend to closely follow the stylus path. In case of a finger, the captured hover events are scattered over a wider area and are rarely directly over the click points.*

A simple straw-hat solution would be to determine the click position only based on the position of the first post-click hover event. However, while this approach works well with stylus clicks, it is inaccurate in estimating finger clicks. The reason is that the stylus, having a smaller pointing surface, generates hover events which tend to follow the trajectory of user movement (Figure 3.3). As a result, the first post-click hover events tend to be very close to the corresponding clicked position. Conversely, the surface of the finger is considerably larger than that of the stylus pointer, and those hover events do not precisely follow the trajectory of the finger movement, as in the stylus case. This was also confirmed by our initial experiment results that showed that the position of the first captured post-click hover is rarely over the position of the click itself.

For this reason, to improve the accuracy of click inference of our approach we use machine learning tools which consider not only the first post-click hover event, but all those captured in the 70ms of the activation of the overlay (see Figure 3.4 for a full attack overview). In particular, for the general input inference attack we use a regression model. For the attacks related to the keyboard (key inference) we make use of a classifier. On a high level, given the set of post-click captured hover events ($h_1, h_2, \ldots, h_n$), a regression model answers the question: *"Which is the screen position clicked by the user?"*. Similarly, the classifier outputs the estimated key pressed by the user. To evaluate our attack we experimented with various regression and classifier models implemented within the analyzer component of the attack using the scikit-learn [144] framework. We report the result in the next section.

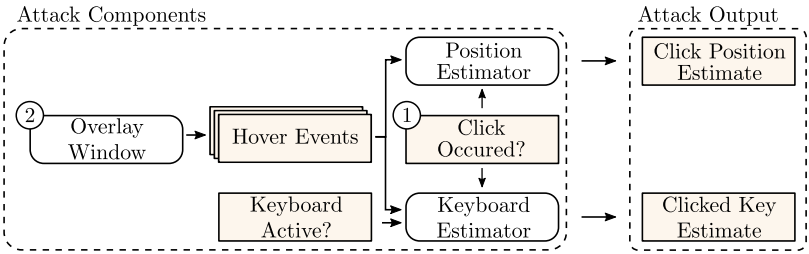


Figure 3.4: *Overview of the attack. Overlay windows catch hover events and detect when a click has occurred. The hover events are then provided to the position estimator regression model which produces an estimated x and y coordinates of the click position. In case of an on-screen keyboard attack, a separate classifier estimates which key was clicked.*

In our initial experiments, we noticed that different users exhibit different hover event patterns, as some users moved the input devices faster than others. In case of fingers, the shape and size of the users' hands resulted in significantly different hover patterns. To achieve accurate and robust click predictions, we therefore need to train the regression and classifier models with data from a variety of users. For that purpose, we performed two user studies that we describe in the next section.

3.4 Attack Implementation and Evaluation

To evaluate the attack we implemented *Hoover*, a prototype malware for Android. Hoover operates in two logically separated steps, as it first collects hover events (as described in Section 3.3) and analyzes them to predict user click coordinates. We implemented the two steps as two distinct components, but both components could also run simultaneously on the user device. However, in our experiments we opted for their functional split as it simplified our analysis. The hover collecting component was implemented as a malicious Android app that runs on the user device. The analyzer was implemented in Python and runs on our remote server, but could also be implemented as another on-device component. The communication among the two is made possible through the `INTERNET` permission held by the malicious app, a standard permission that now Android grants by default to all apps requesting it, without user intervention.

We found that uploading collected hover events on the remote server does not incur a high bandwidth cost. For example, we actively used a

Device Type	Operating System	Input Method
Samsung Galaxy S5	Cyanogenmod 12.1	Finger
Samsung Galaxy Note 3 Neo	Android 4.4.2	Stylus

Table 3.1: *Specifics of the devices used in the experiments. The last column shows the input method supported by the device.*

device (see Table 3.1) for 4 hours, during which our malicious app collected events. The malware collected hover events for approximately 3,800 user clicks. The size of the encoded hover event data is 40 Bytes per each click and the total data to be uploaded amounts to a modest 150kB. We obtained this data during heavy usage of the device and the numbers represent an upper bound. We believe that the average amount of clicks collected by a standard user would be significantly less in a real-life usage scenario.

Finally, for the experiments we recruited 40 participants whose demographics we present in Table 3.2. The evaluation of Hoover was done in different attack scenarios, namely (1) a general scenario, in which we assume the user is clicking anywhere in the screen and two more specific scenarios targeting on-screen keyboard input of (2) regular text and (3) random alphanumeric and symbol strings. We evaluate Hoover in terms of click coordinates inference accuracy in all three scenarios.

We performed a number of experiments with both stylus and finger as inputs, on two different devices whose specifics we shown in Table 3.1. However, the ideas and insights on which Hoover operates are generic and do not rely on any devices particularities. Therefore, we believe our attack would work just as well on other Android devices that support hover.

3.4.1 Use-cases and Participant Recruitment

In this section we describe in detail each use-case scenario and we report on the participants recruited for the evaluation of our attack.

Use-case I: Generic clicks. The goal of the first use-case scenario was to collect information on user clicks anywhere on screen. For this purpose, we asked the users to play a custom game where the users had to recurrently click on a ball shown at random positions on the screen. This use-case scenario lasted 2 minutes.

	Gender		Education			Age			Total
	M	F	BSc	MSc	PhD	20-25	25-30	30-35	
Use-case I	15	5	3	5	12	7	7	6	20
Use-case II	15	5	3	5	12	7	7	6	20
Use-case III	13	7	5	4	11	6	9	5	20

Table 3.2: *Demographics of the participants in our experiments.*

Use-case II: Typing regular text. The second scenario targeted on-screen keyboard input, where the participants were instructed to type a paragraph from George Orwell's book "*1984*". On average, each paragraph contained 250 characters of English text, including punctuation marks.

Use-case III: Typing password-like strings. This time the users were asked to type strings of random characters, instead of English text. Each string had a length of 12 characters that contained combinations of symbols and alphanumeric keys. The goal was to simulate a scenario in which the user is typing a password. An example string looked as follows: "`&!c$$/7#/$;#`". Every time the users had to type a symbol character, they had to click the `SYMBOL` key to switch to the second layout of the keyboard, and vice-versa.

Each use-case scenario was repeated 3 times by the participants. In the first iteration they used their thumb finger as input device. In the second iteration they used their index finger, whereas in the third and last one, the stylus. During each use-case and corresponding iterations we recorded all user click coordinates and hover events that followed them.

Participant recruitment. For the experiments, we enrolled a total of 40 volunteers from a university campus, and we present the demographic details of our participants in Table 3.2. It is worth noting that no private user information was collected at any point during the experiments. The initial 20 people were enrolled for the first two use-cases; the general on-screen click setting and the input of regular English language text. Later on, we decided to add the third use-case to the evaluation as well. We therefore enrolled another 20 volunteers that carried out the corresponding experiments. However, we payed attention that their profile was similar to the participants in the first group (see Table 3.2). The users operated the devices from our testbed (see Table 3.1), with the Hoover malware running in the background. Our set of participants mainly includes the younger

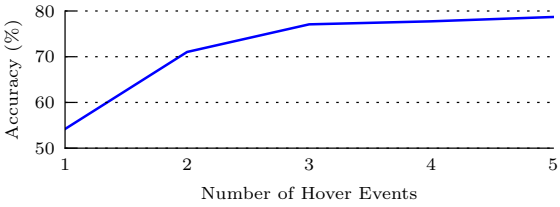


Figure 3.5: *Hover events and accuracy. The input-inference accuracy in dependence of the number of post-click hover events considered.*

population whose input will typically be faster; we therefore believe that the Hoover accuracy might only improve in the more general population. We leave this extended evaluation as part of our future work.

As a result of our experiments with the 40 participants, we collected approximately 24,000 user clicks. Furthermore, the malware collected hover events for 70ms following each click. Approximately 17,000 clicks were of various keyboard keys, while the remaining 7,000 clicks were collected from users playing the ball game.

Ethical considerations. The experiments were carried out by lending each of the volunteers our own customized devices. In the password-like use-case scenario, we asked the participants to type strings with symbols and alphanumerical characters that we randomly pre-generated. At no point did we require participants to use their own devices or provide any private or sensitive information such as usernames or passwords.

3.4.2 Duration of Hover Collection

A first aspect we investigated is for how long the malware should keep the malicious overlay active without obstructing the next click issued by the user. The results showed that, in 95% of the cases, the inter-click time (interval among two consecutive clicks) is larger than 180ms.

Then, we investigated on how the number of post-click hover events impacts the prediction accuracy. For this reason, we performed a preliminary experimental study with just two participants. The initial results showed that the accuracy increases with increasing the number of considered hover events. However, after the first 4 events the accuracy gain is less than 1% (see Figure 3.5). Therefore, for the evaluation of the Hoover prototype we choose to exploit only 4 post-click hoover events. This choice impacted the

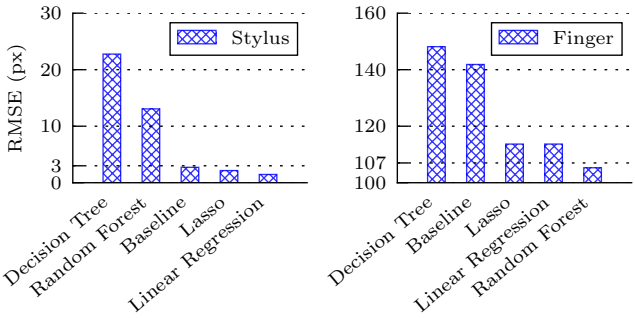


Figure 3.6: *Predicting click positions. We present the results of different regressions models using Root Mean Square Error (RMSE) as the metric. Results are obtained using leave-one-out cross-validation.*

time that Hoover keeps the malicious overlay active for, i.e., the post-click hover event collection time. We observed that 70ms was indeed sufficient, as the first 4 post-click hover events were always fired within 70ms after the user click.

Lastly, we point out that our choice of 70ms is quite conservative when compared with the 180ms of inter-click time observed in our experiments. However, as we will see in the next sections, the prediction results with the Hoover prototype are high. On the one hand, a longer collection time would increase the number of post-hover events captured, and this could improve the accuracy of the regression and classifier in inferring user input. However, a static, longer collection time risks exposing the adversary to users whose click speed is very high — higher than those of the users in our experiment. A more sophisticated adversary could begin with an arbitrarily short collection window and dynamically adapt it to the victim's typing speed.

3.4.3 Hoover Inference Accuracy

In this section, we present the experimental results regarding Hoover effectiveness and precision in inferring user click coordinates. Once Hoover obtains the post-click hover events from the user, it sends them to the machine-learning based analyzer running on the remote server (Section 3.3.5).

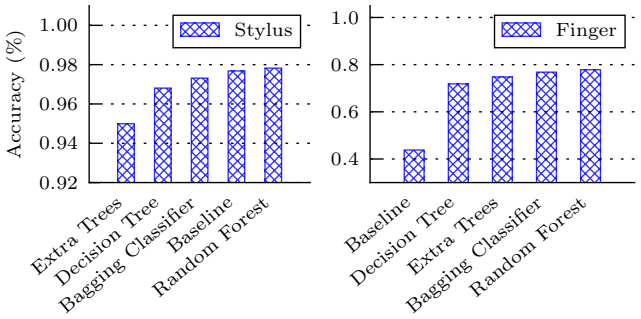


Figure 3.7: *Accuracy in predicting the keyboard keys clicked by the user. The best model (random forest) achieves 98% (stylus) and 79% (finger) accuracy using 10-fold cross-validation. The standard deviation of the values with all models is* $\leq 1\%$.

Inferring the coordinates of general user clicks. The analyzer uses a regression model to infer the user click position on screen. Intuitively, the accuracy of the results depends on the model used for the prediction, and we therefore experimented with a number of different models. In particular, we used two linear models (Lasso and linear regression), a decision tree, and an ensemble learning method (random forests) [144].

The input to each model was, for every subject (user) and click, the (x, y) coordinates of the post-click hover events captured by Hoover (Section 3.3). The output consists of the coordinates of the predicted click position. As a benchmark, we exploit a simple straw-man strategy that outputs the coordinates of the first post-click hover event observed as the estimated click position.

We used the leave-one-out cross-validation, i.e., for every user click validated the training was done on all remaining user clicks. The prediction result for all stylus and finger click samples obtained from the 40 participants in the experiment are presented in Figure 3.6a. We observe that the various regression models perform differently in terms of Root Mean Square Error (RMSE).

First, we observe that for all regression models the finger-related results are less accurate than the stylus related ones. This is expected, as the hover detection technology is more accurate with the stylus (the hover events follow its movement more faithfully) than with the finger (its hover events

are more scattered over the device screen). Nonetheless, in both cases the prediction works well. In particular, we note that the estimation error with the stylus drops down to just 2 pixels. Consider that the screen size of the Note 3 Neo, the smallest device used in the experiments, is 720 × 1280px.

Lastly, we note that in the stylus case (Figure 3.6a) simple linear models perform better than more complex ones. This is not the case when the finger is used as an input device (Figure 3.6b). In this case the best predictions are given by the more complex Random Forest model, which is followed by linear regression. We believe that this is again due to the highest precision with which stylus hovers are captured by the screen, in contrast to those issued by the finger.

Inferring on-screen keyboard input. To infer the typed keys in the keyboard-based use-cases we could follow a straightforward approach. First, infer the corresponding click coordinates with the previous methodology, check into which key locations do the clicks fall into, and then output that key as the prediction result.

Similarly to before, the click prediction in the stylus case and with the linear regression model results are highly accurate — only a 2px error within the actual click coordinate. Therefore, the above straightforward solution might work well for the sylus. However, the procedure is ill suited for the finger case, where the error to predict the coordinates of the clicks is considerably larger (Figure 3.6). For this reason, we take an alternative approach and pose the question as the following classification problem: "*Given the observed hover events, which keyboard key did the user press?*". Again, we experimented with various classification models; two based on trees (decision trees and extra trees), the Bagging Classifier, and the Random Forest model [144]. Similarly to the regression case, we use a baseline model as a benchmark that directly transforms the coordinates of the first post-click hover event into the key they correspond to (whose area they fall within). The results are presented in Figure 3.7 for both the stylus and the finger.

First, we observe that key-prediction results are accurate — 79% for the finger and up to 98% for the stylus (Figure 3.7). The random forest model is the one with the highest prediction accuracy. Additionally, the baseline approach yields an accuracy of 97% with the stylus, the more precise input device in terms of hover events, and the random forest model increases the result by an additional 1%. However, the performance gap between the baseline and the more complex model increases significantly in case of the finger — from 40% (baseline) to 79% (random forest).

3.4.4 Detecting Keyboard Input

Hoover collects any kind of user input that is based on screen clicks. It therefore needs to differentiate among on-screen keyboard taps and other types of clicks, and one possible way is through side channels. Previous work [42] showed that the public `/proc` folder is a reliable source of information to infer the status of other running apps. As the on-screen keyboard is a separate app on Android, Hoover could employ similar techniques to understand when the user is typing. However, we cannot only rely on the approach that uses `/proc` for keyboard detection for two reasons. Firstly, it is not fully accurate and it presents both false positives and negatives, which might diminish our attack's accuracy. Secondly, we cannot be sure that the `/proc` folder will always be freely accessible as the operating system might restrict its access through a specific permission, or even remove it completely for security purposes.

Therefore, we implement a simple heuristic for this problem that exploits the fact that the on-screen keyboard is shown at the bottom of the device screen. Therefore, when a user is typing, the clicks are mostly directed towards the small screen area covered by the keyboard. A straightforward methodology is to employ an estimator to distinguish, among all user clicks, those that target keyboard keys. Such an approach could however result in false positives, as a user could click on the lower part of the screen for many purposes, e.g., while playing a game that involves clicks, to start an app whose icon is located in that area, and so on.

To filter out clicks that could yield false positives we further refine our heuristic. If the user is actually typing, it will issue a large number of consecutive clicks on the lower part of the screen. So, we filter out click sequences that produce text shorter than 4 characters, as these sequences are commonly too short to be usernames or passwords. In addition, we empirically observed that, after the user clicks on a textbox to start typing, at least 500ms elapse until the user types the first key. This is the time needed by the keyboard service to load it on the screen, and we added the corresponding condition to our heuristic to further reduce false positives.

We evaluated the simple and refined heuristic on data gathered for 48 hours from a phone in normal usage, i.e., chatting, browsing, calling, etc. The data consist of clicks and their respective events fired by the system, as well as timestamps of the moments when the user starts (and stops) interacting with the keyboard. Both heuristic versions have no false negatives (missing typed chars) in our dataset. The simple version has a false positive rate of 14%, whereas the refined version reduces it to 10%.

We implemented the heuristics as a proof-of-concept, and we believe a more sophisticated refinement that also includes the difference between typing and clicking touch times (e.g., for how long the user leans the input device on screen during clicks) could lower the false positives further.

3.4.5 Launching Attack Without Any Permissions

In our first proof-of-concept implementation of Hoover we exploited the `SYSTEM_ALERT_WINDOW` permission to generate window views to infer click times (the 0px size Listener view) and to collect post-click hover events (the transparent overlay). However, we show that functionalities of the listener and transparent overlay can be implemented in an alternative, though more complex, way through APIs and functionalities publicly accessible in Android and without requiring any permission at all.

The functionality provided by the listener view (inferring the click time) can be achieved in two different permission-free ways. The first is to analyze information coming from sensors such as gyroscope and accelerometer [134], accessible without permissions on Android. The second is to continuously monitor the information in the `/proc` folder related to the keyboard process, also accessible without permissions on Android [107]. Previous works have shown that both methodologies can be highly accurate in inferring click times [107, 134].

The functionality of the transparent alert view overlay (to capture post-click hover events) can be implemented through the `Toast View` API[1]. `Toast` windows are members of the `View` class which can be used by services without permission in order to provide the user with a quick message regarding some aspect of the system. An example is the window that shows the volume control while the user is changing the volume up or down. Just like alert windows, toasts are always shown on top of the foreground app and they can capture hover events as well. Therefore, the transparent overlay can also be generated as a transparent `Toast View` that covers the whole screen after each user click, and can be used to capture the necessary post-click hover events.

As a proof-of-concept, we implemented a preliminary version of Hoover with the above insights. In particular, we used side-channel information from the `/proc` folder to infer user click times and the Toast APIs for the transparent overlay. We evaluated this permission-free version of the attack over data collected through a small-scale experiment — a single-user instructed to perform the first two use-case scenarios (clicking on a

[1] `https://developer.android.com/guide/topics/ui/notifiers/toasts.html`

ball and typing regular text) with both finger and stylus as input device. Even with this small-scale evaluation the accuracy was high, as the random forest classifier yielded a 92% accuracy in inferring user clicks, while the regressor had an RMSE of 112px. These results show that Hoover can be implemented without relying on *any permission* and still be highly accurate.

3.4.6 Attack Improvements

The experimental results presented in this section showed that hover events can be used to accurately infer user input, both in case of general click as well as keyboard keys. In this section we list other ways that could potentially improve the attack and its accuracy even further.

Additional hover events. Throughout our evaluation, we collected 4 hover events following each click. An attacker could collect more hover events, for a period of time longer than 70ms, which could further improve the performance of our models.

Language model. In our evaluation, we considered the worst case scenario for the attacker, where the attacker does not make any assumptions on the language of the input text. Although the text the users typed was in English in our experiments, it could have been in any arbitrary language. In fact, the goal of the experiment was just to collect data on user clicks and hover events, irrespective of the typing language. A more sophisticated attacker could therefore first detect the language the user is typing in. Then, after the key inferring methods we described, the attacker could apply additional error correction algorithms to improve the accuracy.

Per-user model. In our evaluation, both the regression models and classifiers were trained on data obtained from all users, i.e., for each strategy we created a single regression and classification model that was then used to evaluate all users. However, it is reasonable to assume that a per-user model could result in higher accuracy. We could not fully verify this intuition on our dataset as we did not have long enough per-user data on all participants. However, we did a preliminary evaluation on the two users with the most data points: 411 clicks for user 1 and 1,399 for user 2. The result with separate per-user model training showed a considerable improvement, particularly with the finger typed input. The accuracy of keyboard key inference increased from 79% (all users) to 83% for the first user and 86% for the second one.

Alternative input methods. Our attack is both effective and accurate in keystroke inference, but it can not be as effective with swiping text. As Hoover infers coordinates only after the input device leaves the screen, in case of swiping this translates to inferring only the last character of each word swiped by the user, which does give enough information on the typed text. That said, it is important to note that swiping is not enabled for password-like fields and characters such as numbers or symbols need to be typed and not swiped. Therefore, even in the presence of swiping, Hoover is still effective in stealing sensitive information such as passwords or pins.

In our attack we assume that a regular keyboard is used. However, users could employ complex security mechanisms that, e.g., customize the keyboards or rearrange the keys each time the user types. This type of mechanisms would certainly mitigate our attack as Hoover would be unable to correctly map coordinates to keys. However, at the same time the usability of the device would decrease as the users would potentially find it difficult to write on a keyboard whose keys are rearranged each time. Consequently, it is likely that systems would tend to restrict the protection mechanism to highly sensitive information like PIN numbers and credentials, leaving texts, emails, and other types of messaging sequences still vulnerable to our attack.

3.5 Attack Implications

The output of our attack is a stream of user clicks inferred by Hoover with corresponding timestamps. In the on-screen keyboard use-case scenario, the output stream of clicks can be converted into keyboard keys, either using our trained classifier or other alternative means (see Section 3.4.4). In this section we discuss possible implications of the attack and of the techniques and ideas exploited therein.

3.5.1 Violation of User Privacy

A first and direct implication of our attack is the violation of user privacy, as a more in-depth analysis of the stream of clicks could reveal sensitive information regarding the device owner. Consider the following output of our attack:

```
john doe<CLICK>hey hohn, tomorrow at noon, downtown
starbucks is fine with me.<CLICK><CLICK>google.com
<CLICK>paypal<CLICK>jane.doe<CLICK>hane1984
```

After an initial analysis of the sequence we quickly understand that the first part of the corresponding user click operations were to either send an email or a text message. We also understand who is the recipient of the message (probably John) that the user is meeting with the person the next day, and we uncover the place and the time of the meeting. Similarly, the second part of the sequence shows that the user googled the word `paypal`, and that the user probably logged into the service afterwards, that her name is Jane Doe and that her credentials for accessing her `paypal` account are potentially `jane.doe` (username) and `jane1984` (password). This is a simplified example that shows how easily Hoover, starting from just a stream of user clicks, can infer sensitive information about a user.

Another observation in the above example is that the output contains errors regarding the letters "`j`" and "`h`" as the corresponding keys are close on the keyboard. However, since the text is in English, simple techniques based on dictionaries can be applied to correct for the error. If the text containing the erroneous inferred key was a password, dictionary based techniques would not work as well. However, in such cases we can exploit another aspect; namely the movement speed, angle, and other possible features that define the particular way each user moves the finger or stylus to type on the keyboard. It is likely that this particularity impacts the key-inference accuracy of Hoover and that makes so that a specific couple of keys, like "`j`" and "`h`", tend to be interchanged. With this in mind, from the example above we can deduce that Jane's password for the `paypal` account is likely to be `jane1984`.

3.5.2 Advanced Analysis of Target Applications

There are other, more subtle, potential uses of the sequence of user input events that Hoover collects. For example, from the click streams the adversary could potentially uncover the foreground app (the one the user is currently interacting with). This could be done by inferring which app icon was clicked on the main menu of the device or by fingerprinting the interaction between the user and the target application. Indeed, every application could be potentially associated to its unique input pattern. Once the foreground app is known, the adversary can launch other, more damaging attacks that target the particular application through attacks like UI redressing or phishing.

3.5.3 User-biometrics Information

So far we have discussed what an adversary can obtain by associating the user click streams inferred by Hoover to their semantics (which app was

started, typed text, etc.). However, the data collected by Hoover could also be used to profile the way a user clicks or types. As Hoover has access to click timestamps, it could therefore deduce user biometric information regarding device interaction patterns.

The listener view in Hoover obtains the corresponding timestamp each time a hover event is fired in the system. In particular, it obtains timestamps for events of the type *touch down* (the user clicks) and *touch up* (the user removes the input device from the screen). These timestamps allow Hoover to extract the following features: (1) the click duration, (2) the duration between two consecutive clicks, computed as the interval between two corresponding *touch down* events, and (3) hovering duration between two clicks, computed as the interval between a *touch up* event and the next *touch down* event. These features are the fundamentals for continuous authentication mechanisms based on user biometrics [153,195]. In addition, the mechanisms proposed in [153, 195] require a system level implementation, which can be challenging and may add complexity to existing systems. Hoover could therefore be used to attack biometric-based authentication mechanisms as well.

3.6 Discussion and Countermeasures

The success of the attack we described relies the a combination of an unexpected use of hover technology and alert window views. Here we review possible countermeasures against this attack and we show that, what might seem like straightforward fixes, either cannot protect against the attack, or severely impact the usability of the system or of the hover technology.

Limit access to hover events. The presented attack exploits the information dispatched by the Android OS regarding hover events. In particular, the hover coordinates can be accessed by all views on the screen, even though they are created by a background app such as Hoover. This feature enabled us to accurately infer user input. One possible way to mitigate the attack is to limit the detection of hover events only to components generated by the application running in the foreground. In this way, despite the presence of the invisible overlay imposed by Hoover running in the background, the attacker would not be able to track the trajectory of the movement while the user is typing. However, this restrictive solution could severely impact the usability of existing apps that use alert windows to improve the overall user experience. An example is the “chat head” feature of the Facebook application. If not enabled to capture hover events, this feature would not

serve it's intended purpose as it would not capture user clicks either. Recall that a view either acquires both clicks and hover events, or none of them.

Another possibility would be to decouple hover events from click events, and limit the first ones only to foreground activities. This solution would add complexity to the hover-handling components of the system and would require introducing and properly managing additional, more refined permissions. Asking users to manage complex permissions was shown to be inadequate, as users tend to agree to any permission requested by a new app they want to install [117]. Not only users, but developers as well find already existing permissions as complex and tend to over-request permissions to ensure that applications function properly [62]. Given this, introducing additional permissions to address this problem is challenging.

Use the touch filtering specific. We proceed by explaining why the `filterTouchesWhenObscured` mechanism [4] cannot be used to thwart our attack. Touch filtering is an existing Android OS specific that can be enabled for a given UI component. When enabled, all clicks (touch events) issued over areas of the view obscured by another service's window, will not get any touch events, i.e., the view will never receive notifications from the system about those clicks. The touch filtering is typically disabled by default, but app developers can enable it for components, including views, of a given app by calling `setFilterTouchesWhenObscured(boolean)` or by setting the `android:filterTouchesWhenObscured` layout attribute.

If Hoover were to obstruct components during clicks, the touch filtering could have endangered its stealthiness as the underneath component to whom the click was intended to would not receive it, so the user could become alerted. However, this is not the case, as Hoover never obstructs screen areas during clicks. Recall that, the malicious overlay is created and destroyed in appropriate instants in time, as to not interfere with user clicks 3.3. Therefore, even with the touch filtering enabled by default on every service and app, neither the accuracy, nor the stealthiness of the Hoover malware are affected.

Forbid 0px views. Hoover uses a 0px view which listens for on-screen touch events and notifies the malware about the occurrence of a click so it can promptly activate the transparent overlay. Thus, forbidding the creation of 0px views by services seems like a simple fix to prevent the attack. However, the attacker can still overcome the issue by generating a tiny view and position it on the screen as to not cover UI components of the foreground app. For example, it could display it as a thin black bar on

the bottom, thus visually indistinguishable from the hardware screen border.

Limit transparent views to legitimate or system services. Clearly, this limitation would impede Hoover to exploit the transparent overlay. However, a sophisticated attacker can employ a more complex technique to overcome the problem. For example, in the keyboard attack scenarios, Hoover can, instead of showing a transparent overlay, design a non-transparent one which is an exact copy of the keyboard image on the victim's phone. The keyboard-like overlay would then operate just as the transparent one yet would be equally undetectable to the user. A similar approach can be used to collect the clicks of a target app whose design and components are known to the attacker (e.g., the login screen of well-known apps such as Gmail, Facebook, and so on).

Inform user about overlays, trusted paths. The idea here is to make the views generated from a background service easily recognizable by the user by restricting their styling, e.g., imposing at system level a well-distinguishable framebox or texture pattern. In addition, the system should enforce that all overlay views adhere to this specific style, and forbid it for any other view type. However, countermeasures that add GUI components as trusted path [31, 183] to alert a user about a possible attack (security indicators) have effectiveness issues. Even when users were aware about the possibility of an attack and the countermeasure was in place, there were still 42% of users that kept using their device normally [31].

This kind of trusted paths mostly help against phishing attacks where the foreground app is a malicious one and not the actual app the user intends to interact with. However, note that this is not the case for our attack where the malware always runs in the background and does not interact with the legitimate foreground application. Even if security indicators were shown on a view-based level rather than on an app-based one like in [31], note that the overlay in Hoover is not static. Rather, it is shown for very short time windows (70ms) successive to a click, when the user focus is potentially not on the security indicator.

Protect sensitive views. Another idea is to forbid that a particularly sensitive view or component generated by a service, e.g, the keyboard during login sessions or an install button of a new app, is overlaid by views of other services, including alert windows. A possible implementation of this solution could be to introduce an additional attribute of the `View` class, which specifies whether a given instance of the class should be "coverable".

When this attribute is set, the system could enforce any other screen object overlapping with it to be "pushed out" the specific view's boundaries, e.g., in another area on the screen not covered by the view. However, it would be a responsibility of the app developers to carefully design the app and identify sensitive views that require the non-coverable attribute. In addition, these types of views should adhere to a maximum size and not cover the whole screen. Otherwise, it would not be possible for other services, including system ones, to show alert windows in presence of a non-coverable view. This solution could mitigate attacks like ours and also others that rely on overlays even though in a different way (e.g., clickjacking). However, it would put a considerable burden on the app developers that will have to carefully classify UI components of their apps into coverable and non coverable, taking also into consideration possible usability issue with views generated unexpectedly from other legitimate apps such as on screen message notifications, system alert windows, and so on.

Restrict access to trusted applications and services. Finally, Android could restrict apps from accessing features of the system that could be exploited in attacks at system level, rather than leaving the final decision at the hand of the user or developers. This approach is partially adopted by iOS for certain sensors (e.g., the microphone), which is limited only to apps that require it to function correctly. Another possibility is to grant the `SYSTEM_ALERT_WINDOW` permission only to system services, or to apps signed by the Android development team [17]. However, this solution is costly and might require manual intervention but, nonetheless, combined with user alerts could largely mitigate the attack.

From the discussion above, we conclude that access to hover events needs to be handled carefully, taking into consideration both usability and security concerns. However, it should not be ignored since it allows input inference at a very high level of granularity and with a very high accuracy.

3.7 Related Work

The main challenge to achieve the goal of inferring user input comes from the basic rule of Android, that a click is only captured by one app only. However, existing works have shown that malware can use various techniques to bypass this rule and infer user input (e.g., steal passwords).

We can think of mobile application phishing [42] as a trivial case of input inference attacks, where the goal of the malware is to steal keyboard input (typically login credentials) of the phished application. Although effective

when in place, a limitation of phishing attacks is their distribution. To spread the malware, the attacker commonly submits the app to official markets. However, markets can employ stringent checks on the apps and perform automated app analysis. Furthermore, and contrary to our techniques, phishing attacks need to be implemented separately for every phished app.

UI redressing (e.g., clickjacking) is another approach to achieve input inference on mobile devices [31, 89, 139, 150, 152, 191]. These techniques operate by placing an overlay window over some component of the application. When clicked, the overlay either redirects the user towards a malicious interface (e.g. a fake phishing login page), or intercepts the input of the user by obstructing the functionality of the victim application (e.g., an overlay over the whole on-screen keyboard). However, such invasive attacks disrupt the normal user experience, as the victim application never gets the necessary input, which can alarm the users.

An alternative approach is to infer user input in a system-wide manner by using side-channel data obtained from various sensors present on the mobile platform [68, 85, 132, 134, 137, 161, 178, 196], like accelerometers and gyroscopes. Reading such sensor data commonly requires no special permissions. However, such sensors provide signals of low precision which depend of environmental conditions (e.g. the gyroscope of a user that is typing on a moving bus). The derived input position from such side-channels is therefore often not accurate enough to differentiate, e.g., which keys of a full on-screen keyboard were pressed. For example, the microphone based keystroke inference [137] works well only when the user is typing in portrait mode. In addition, its accuracy depends on the level of noise in the environment.

Contrary to related works, our attack does not restrict the attacker to a given type of click-based input (e.g., keyboard input inference only), but targets all user clicks. It does not need to be re-implemented for every target app, like phishing and UI redressing, as it works system-wide.

3.8 Conclusion

In this chapter, we proposed a novel type of user input inference attack. We implemented *Hoover*, a proof-of-concept malware that records user clicks performed by either finger or stylus as input device, on devices that support the hover technology. In contrast to prior works, our attack records all user clicks with both high precision (e.g., low estimation error) and high granularity (e.g., at the level of pressed keyboard keys). Our attack is not tailored to any given application, and operates in a system-wide manner.

Furthermore, our attack is transparent to the user, as it does not obstruct normal user interaction in any way.

Attack limitations. The current limitation of hover technology, as well as our attack, is the inability to detect multiple input devices (e.g., fingers) at the same time. However, we believe that, as soon as multi-hovering is implemented on mobile devices, the attacks presented in this chapter could be adapted. An additional limitation is that our attack is not tailored for alternative input methods such as swipe.

Chapter 4

Adaptive User Interface Attacks

4.1 Introduction

In the previous chapter, we proposed a novel input inference attack whose goal was to compromise the confidentiality of user input, and our previous attack required a malicious application to run on the target device. However, modern malware can reside in various places — on the device itself, but also on connected peripherals (e.g., BIOS, hard drive firmware). One type of such malware resides in malicious user interface devices, such as a keyboard or a mouse [140]. Commonly, the end goal of such malicious peripherals is to inject sequences of user input that install some form of malware to the device operating system. However, installing malicious OS components or introducing system misconfigurations, such as adding an administrative account, could be fully prevented by security hardening (e.g., Windows Embedded Lockdown features [133]), or leave forensic traces that can be detected by existing malware-detection approaches.

A significantly stealthier alternative, that after attack completion leaves little to no forensic evidence, is to attack systems *only* through their user interfaces. Such attacks exploit a fundamental property of any device designed to operate under user control; namely that, irrespective of applied hardening techniques, the device must continue to accept user input.

As user input cannot be simply blocked, user interface (UI) attacks were proposed. Current state-of-the-art are BadUSB-style attacks [140], where

the goal of the malicious peripheral is to inject simple, pre-programmed sequences of keyboard and mouse input, commonly while the user is not using (or looking at) the device. However, the main drawback of such works is the lack of system state awareness. For example, the BadUSB malware does not know in which state the system is currently in. Launching attacks that inject user input at the wrong time could result in attack failure, or in users trivially detecting them. If the malware could infer the system state, and the precise point in time when to launch an attack, more powerful attacks (e.g., compromise integrity of e-banking payment), as well as new attack scenarios, become possible. We therefore attempt to answer the following question: *"Can an adversary improve state-of-the-art UI attacks, without installing malware on the target system?"*

We present a new class of adaptive runtime user interface attacks, in which the adversary infers the system state, violates the integrity of user input at a specific point in time, *while the device is operated by the legitimate user*, causing precise and stealthy runtime UI attacks without any malware running on the device. The attack is hard to detect, it can result in serious safety violations, even loss of human life, and users are led to believe that they accidentally caused the damage themselves.

The first part of the attack is conventional. The adversary gains temporary physical access to the target system, attaches a small attack device, and leaves the premises. The device is attached to an interface that connects an input device (USB keyboard or mouse, touchscreen, etc.) to the system. After deployment, the attack device works fully autonomously, i.e., without remote connection to the adversary.

In the second part of the attack lies its novelty. Contrary to existing approaches, our attack device observes the constant stream of user input events and, based on this information, determines when the user is about to perform a critical operation, and when the UI attack should be launched. Although the attack device has full visibility and control of the user input channel, it *cannot directly observe the state of the target system*, as the device has no feedback from the system (e.g., access to monitor output). In particular, the adversary does not know the current state of the UI or the mouse pointer location. The adversary must therefore, given user input, infer the most likely system state and correct attack timing. We propose the following two attack approaches:

- *UI fingerprinting*. If the attack user interface is sufficiently unique in it's design (e.g., number, type and positions of UI elements), then the attacker can precisely detect when the user is interacting with that

Pacemaker Programmer

Touchscreen and Joystick Operated Robot

E-Banking Websites

Figure 4.1: *Examples of critical user interfaces on dedicated terminals and general-purpose PCs. Attacking such UIs can lead to various safety and security violations.*

UI by simply observing the mouse and keyboard input patterns that are unique to it.

- *State tracking*. If the adversary can build a model of the entire target system user interface, the adversary can attack the system using probabilistic state tracking.

To successfully realize our UI attacks, we had to overcome technical challenges. Once the adversary has determined the correct time to attack, the attack device injects a series of precise and fast input events. While the adversary is able to freely manipulate the input channel, the user receives instant visual feedback — the legitimate user is part of the system control loop. Therefore, we designed attack techniques that are both accurate (low false positives), and stealthy (give little visual indication to the user). Furthermore, our attack needs to run in small attack devices with constrained resources, track system state both accurately and fast.

To demonstrate the attack, we implemented it on a small embedded device, and evaluated it on two different types of platforms: general-purpose PCs and dedicated terminals.

First, on PC platforms, we tested our attack on replicated UIs of real-world e-banking websites from three major banks, on 20 local participants. Each participant was a domain expert, as the attack was performed using the bank they were already a client of, on a UI they were already familiar with. Our evaluation shows that we can accurately fingerprint UIs in a reliable (90% attack success rate) and stealthy (90% of users did not notice our attack) manner, that is surprisingly tolerant to noise (users habitually clicking around, pressing “tabs” to navigate between elements, different browsers and screen resolutions).

Second, for dedicated terminals, we tested the attack on a simulated (but realistic) user interface of a medical implant programmer. We tested our attack on 987 online study participants and we show that the we can accurately perform system state tracking, and that the attacks are very hard to detect (93% to 96% attack success rate). To gain increased confidence in our online study results, we performed a follow-up, on-site study with domain experts: none noticed the attack.

We emphasize that our attack approach is applicable to a wide range of attack scenarios, including different target systems, user input methods and UIs, where the adversary has to perform the attack under target state uncertainty. We show that our novel UI attack approach is easy to deploy, as it requires only brief and non-invasive access to the target system (e.g., attaching a USB device takes seconds). A small attack device (e.g., NSA cottonmouth project[1]) can be difficult to notice, and to a malware detection system the attack is invisible, as no malicious code is running on the terminal itself. The attack is also agnostic to any applied user authentication.

One way to detect the attack is that the user notices the subtle visual changes on the user interface while the attack is active (e.g., medical device settings are modified). However, our user studies show that the vast majority of users fail to notice the attack. The prevention of this attack therefore likely requires different approaches, such as authentication of input devices, design of protective measures on user interfaces, etc., and this work motivates the development of such solutions. Since our attack is invisible to traditional malware detection, it operates under uncertainty without any feedback from the system, and it gives little visual indication to the user, we call it *hacking in the blind*.

4.2 Background on Terminals

In this section we give background on terminals and their security hardening, and motivate that terminals can be susceptible to the types of UI attacks that we focus on in this chapter.

Many safety-critical systems are embedded terminals that run stripped-down operating systems with hardened security. Here we describe protective measures (*Lockdown features*) available on the Windows Embedded Industry platform [133], but similar security enhancements are commonly available on embedded Linux distributions as well.

The terminal can be configured to run only a single application (AppLocker feature) where the application user interface occupies the entire

[1] `https://nsa.gov1.info/dni/nsa-ant-catalog/usb/index.html`

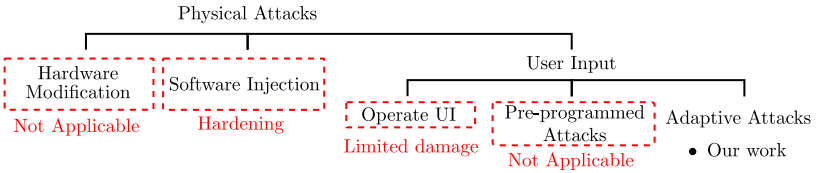


Figure 4.2: *Classification of physical attack techniques and their limitations. Remote attacks are typically not applicable to devices that are, e.g., connected to closed networks or even air-gapped. Hardware modifications are time consuming and require a sophisticated adversary. Runtime software injection (e.g., into the device RAM) can be prohibitively complex, and can leave forensic evidence. Attacks where the adversary operates the device UI typically have limited damage. Pre-programmed user input attacks, such as BadUSB [140], do not work on safety-critical terminals due to missing keyboard shortcuts and console access. The focus of our work is on adaptive user input attacks that overcome these limitations.*

device screen. The user cannot escape the application UI with a specific key sequence (Keyboard Filter), and thus the user can only interact with the terminal through that UI and the application is executed with least user privileges (User Account Control). Terminals are typically disconnected from the Internet, installation of third-party software is not allowed, and the terminal can verify its software at boot and start only signed software at runtime (AppLocker). The terminal can be configured to only connect to USB devices with known class, device and product identifiers (USB Filter).

To understand if our attack is applicable to safety-critical terminals, we studied medical terminals and their deployment in two ways. First, we visited the intensive care unit of a modern hospital and examined its terminals and physical access control practices. We were both surprised and disturbed just how easy it is to get brief, unattended physical access to medical terminals that are used for life-critical operations. Based on our experience, an adversary could easily access medical terminals, and plug a small device to a port that connects an input peripheral.

Second, we performed a survey of 130 medical terminals based on publicly-available manuals. We found that 76 of the studied devices are operated by some combination of mouse, keyboard, or touchscreen. Of those 76, we found accessible USB ports on 50 devices. We conclude that

safety-critical medical terminals with easily accessible, off-the-shelf input peripherals and ports are common. Our survey data is available online[2].

4.3 Problem Statement

The goal of the adversary is to attack a security-critical user interface (see Figure 4.1), and we focus on attack scenarios where the adversary has brief physical access to the target system, prior to its usage. In this section we discuss limitations of known physical and user interface attack techniques and describe our adversary model.

4.3.1 Limitations of Known Attack Techniques

There are various kinds of physical attack. Figure 4.2 provides a categorization of common attack techniques.

Hardware modification. One could argue that, in case the attacker has even brief physical access to a device, that the device should already be considered as trivially compromised. However, this is often not the case, due to two practical reasons. First, the attacker often can not shut the device down. This would prevent the attacker from opening the device and injecting advanced hardware backdoors or performing similar modifications. Second, the attacker may simply not have sufficient time to perform such attacks. For example, in case of the hospital, we observed that the intensive care ward was never left unattended for extended periods of time.

Software injection. Another approach that relies on physical access is local injection of malicious code. Existing terminal hardening techniques prevent simple malicious code installation. The terminals can be configured such that code cannot be copied to terminals from external media (e.g., connected USB devices) and unsigned code cannot be executed on the terminals. More sophisticated code injection techniques that, for example, inject malicious code to the runtime memory of the terminal require sophisticated equipment and are difficult to execute, given only brief physical access to the terminal. On general-purpose PCs, code injection attacks are common, but such attacks leave forensic evidence that could be detected using existing malware-detection schemes.

Operate user interface. User input can not be simply blocked, and an attack approach that leverages this fact is to manipulate the device directly

[2] `https://goo.gl/arp2DU`

through its user interface. For example, if the adversary has physical access to the target device, and can operate its UI, the adversary can perform all the operations that the legitimate user is entitled to. Such attacks have two limitations. First, unauthorized use can be addressed with user authentication (e.g., devices could be screen locked). Second and more importantly, on safety-critical terminals the damage of such attacks is typically limited. For example, certain medical devices are only connected to patients when operated by doctors, and thus such attacks are less severe compared to runtime attacks that modify the operation of the terminal during its use. Similarly, e-banking websites are protected using passwords and second-factor authentication schemes — information that the attacker does not necessarily have access to at attack time.

Pre-programmed attacks. Another class of user input attacks rely on external devices connected to the target system. In such cases, no malware is present on the target system, and the purpose of the attack device is to either passively intercept user input or to actively inject pre-programmed sets of commands. A hardware key logger that collects user input is an example of a passive attack. BadUSB [140] is an example of a pre-programmed attack where a malicious user input device (keyboard) injects malicious keystrokes into the target system. Such attacks leverage keyboard shortcuts that, e.g., open a console or an administrative window and modify the target system settings. Such attacks do not apply to hardened terminals, as they rarely have console programs, and on terminals the user, or an adversary that controls user input, cannot "escape" the application UI to modify system settings beyond what the application enables. As the adversary does not know the current system state, such attacks can not be used to compromise (e.g., hijack) an e-banking user session, without resorting to installing malware to the device.

4.3.2 Our Goal: Adaptive Runtime Attacks

The focus of this chapter is on *adaptive runtime UI attacks*, that overcome the limitations of the above discussed techniques. We explore UI attacks that work even if hardware modifications are not practical and software injection can be prevented. Our goal is to design runtime UI attacks that are more damaging and accurate than pre-programmed attacks or operating the device directly through its user interface.

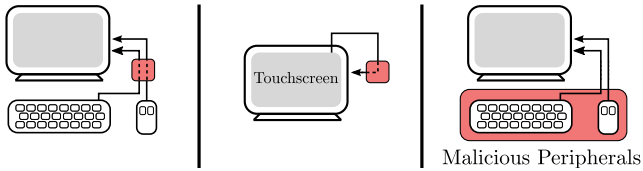


Figure 4.3: *Attack scenario. The adversary attaches an attack device to an interface that connects a user input device to the target device. The adversary can also replace a user input device with a malicious one.*

4.3.3 Adversary Model

We assume an adversary that gains temporary access to the target device prior to device usage, and attaches a small attack device to it and leaves the premises. The attack device sits in-between a user input peripheral and the device (Figure 4.3). If the input peripheral is integrated into the terminal device (e.g., touchscreen), the adversary attaches the attack device to an interface that connects the input device to the terminal main unit. If the input device is an external device (e.g., USB mouse), the adversary can attach the attack device to the USB port that connects the device to the target system. The adversary can also simply replace an external input peripheral with one that contains the attack device. Such on-site operations are quick to perform. General-purpose PCs have ports for user input devices. Additionally, our survey (see Section 4.2) shows that many terminals are used with external peripherals that are connected to easily accessible ports.

We assume that the adversary can install the attack device unnoticed. Besides installing the attack device, the adversary does not interact with the target device in any other way. In particular, we assume that the adversary does not reboot the device and that the adversary cannot observe its current state (the user interface might be locked, or password protected). The adversary can make the attack device so small that legitimate users do not notice its presence. If the device is used via two input devices (e.g., mouse and keyboard), the adversary can connect both to the same attack device.

The attack device can observe, delay, and block all events that originate from the connected user input devices as well as inject new events. After the installation, the attack device works fully autonomously, i.e., it does not communicate with the adversary. After the attack, the adversary may collect the attack device.

We assume that the adversary knows the target application UI, including its states and state transitions. We also assume that the application user interface is deterministic: similar interaction always causes the same result.

4.3.4 Example Attack Targets

We explore such attacks in two different contexts: general-purpose PCs and dedicated terminals. On PC platforms we focus on online banking user interfaces. If the adversary manages to launch a successful UI attack on e-banking, the attack can have significant financial consequences. On terminals we focus on medical implant programmers. If the adversary can successfully violate integrity of user input on an implant programmer UI, the attack can put human lives in danger.

These two platforms, and the chosen application UIs, represent different types of attack targets. The user interface of a dedicated terminal device typically consists of a single application UI. The application UI typically occupies the entire screen and the user of the device cannot escape the application UI. Terminal devices often have fixed screen resolutions. When the terminal is booted, its execution always begins from the same application UI state. On the other hand, general-purpose PCs have many applications with various UIs. The application UIs are managed by windowing systems and users are free to install new applications. The UIs of general-purpose PCs also have various screen resolutions that can be changed by the user. When a PC is booted, the attack target application is typically not active.

4.4 Hacking in the Blind

The installed attack device observes events from the connected input device(s) and launches the attack by modifying existing or injecting new input events when the legitimate user is performing a security-critical operation.

While the attack device can intercept all user input events, their interpretation may have two forms of uncertainty. First, the adversary may not know the state of the target device UI (e.g., because the UI was locked when the attack device was installed). We call this *state uncertainty*. Second, the adversary may not be able to interpret all received user input events without ambiguity. In particular, mouse events are relative to the mouse cursor location that may be unknown to the adversary. We call this *location uncertainty*. In contrast to mouse input, touchscreen events do not have location uncertainty as touchscreen clicks are reported to the operating system in terms of their absolute (x, y) coordinates.

The primary challenge in our approach is to launch the attack accurately under such uncertainty, without any feedback from the target device (hacking in the blind). The best attack strategy depends on the attack platform (general-purpose PC or dedicated terminal), application user interface configuration, the type of the input device, and the level of stealthiness the adversary wants to achieve. We first discuss simple attack strategies, and then move on to present our two main attack techniques: state tracking and UI fingerprinting.

4.4.1 Simple Techniques

If the adversary manages to reduce (or remove) both location and state uncertainty, attacking the device user interface becomes easier. Assuming that the adversary knows both the current user interface state, the mouse cursor location, and complete model of the UI, each event can be interpreted unambiguously. For example, if an adversary knows the complete user interface configuration of a dedicated-purpose terminal, the adversary can then easily track both mouse movement and state transitions in the user interface. In general-purpose platforms, such as PCs, building such a model is typically not possible. Below we list methods that can help the adversary to reduce uncertainty.

Reducing state uncertainty. A simple technique to learn the state of the system is to wait for a reboot. If the attack device can determine when the terminal is booted, it knows that, shortly after, the target device user interface is in a known state. This technique works only if the target device is rebooted before the attack. While PCs are routinely restarted, many dedicated terminals run long periods of time without reboots. Learning the state of the target device at boot may help attack state detection for system where complete UI model can by built. However, on general-purpose PCs, where such models are typically not feasible, knowledge of the correct system state at boot does not help in attack state detection.

Reducing location uncertainty. A simple technique to determine the mouse cursor location is to actively move the mouse (i.e., inject movement events) towards a corner of the screen. For example, if the mouse is moved up and left sufficiently, the adversary knows with certainty that the mouse cursor is located at the top-left corner of the screen. Moving the mouse while the system is idle may not be possible if the target device UI is locked.

To make the above process appear less suspicious during terminal use, the adversary can create an appearance that the user moved the mouse

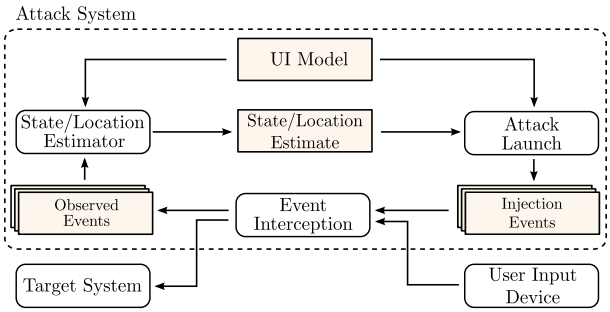


Figure 4.4: *Attack system overview. On the attack device, a state tracking component processes observed user input events with respect to a UI model and produces a state estimate. The attack is launched based on the state estimate and the UI model by blocking and injecting new events.*

herself. For example, when the attack device observes mouse movement events left and up, it can inject additional movement events to the same direction. The process mimics a situation where the mouse movement was accelerated and the user unintentionally moved the mouse cursor to corner of the screen herself. The limitation of this technique is that if such mouse movement is performed repeatedly (e.g., at every boot), it can appear suspicious to an anomaly detection system or post-attack forensics.

Summary. We take a practical stand and assume that in many scenarios the adversary has to perform the attack under location uncertainty, state uncertainty or both. To increase the robustness of our attack, we design attack techniques to handle both types of uncertainty. We first describe state tracking that is applicable to target devices, such as terminals, where the adversary can build a complete model of the target system UI. After that, we describe UI fingerprinting that allows attack state detection on platforms, such as general-purpose PCs, where such model creation is infeasible.

4.4.2 State Tracking

Starting from this section, we describe a novel state tracking that enables the adversary to launch accurate attacks despite of uncertainty. State tracking is applicable to terminals, where the adversary can build a complete model of the target system UI (e.g., typically on dedicated terminals, the target application UI constitutes the complete target system UI). A noteworthy property of the system is that it estimates user interface state and mouse

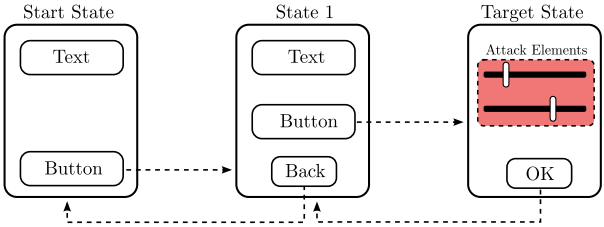


Figure 4.5: *Example user interface model. The model includes user interface states, user input elements and state transitions. The attack state contains attack elements and a confirmation element.*

location *fully passively*, and thus enables implementation of stealthy attacks. We proceed by giving a high-level overview of the attack system (Figure 4.4).

The attack device contains a static model of the target system user interface that the adversary constructed before the device deployment and it runs two main software components. The first component is a *State and Location Estimator* that determines the most likely user interface state (and mouse cursor location) based on the observed user events and the UI model. The estimation process can begin from a known, or an unknown user interface state, at an arbitrary moment and it tracks mouse, keyboard and touchscreen events. The second component is an *Attack Launcher* that performs active UI manipulation when the legitimate user is performing a safety-critical operation. We describe several attack variants and evaluate their detection through user studies.

User interface model. The user interface model (Figure 4.5) contains user interface states, their user input elements and state transitions. User input elements are buttons, text fields, multiple choice elements, sliders etc. All input elements can be interacted with mouse and touchscreen devices, while some can also be interacted with a keyboard. For each state, the model includes the locations and types of the user input elements and the possible state transition that the element triggers. One state is defined as the start state and one or more states are defined as the target states. The goal of the attack is to modify safety-critical input elements (*target elements*) on the target states. Typically the target state includes also a confirmation element that the user clicks to confirm the safety-critical operation.

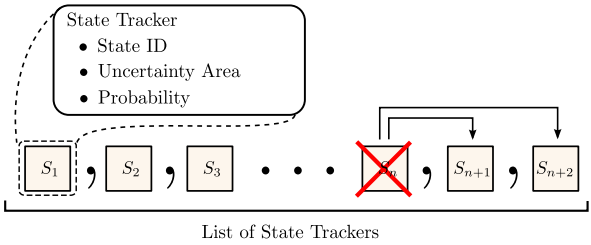


Figure 4.6: *Our algorithm maintains a list of state trackers. On each click it creates child trackers and removes the parent from the list.*

State and location estimation. Here we describe our state and location estimation algorithm for mouse and keyboard. Later we explain how the same algorithm can be used to estimate state for touchscreens input (only state uncertainty).

The algorithm operates by keeping track of all possible user interface state and mouse location combinations. For each possible state and location the algorithm maintains a *state tracker* object. The trackers contain a state identifier and an *uncertainty area* that determines the possible location of the mouse in that state instance. Additionally, the algorithm assigns a probability for each tracker object that represent the likelihood that the terminal user interface and the mouse cursor are in this state and location.

The estimation algorithm maintains the tracker objects in a list (Figure 4.6). If the estimation begins from a known state, we have initially only one tracker, to which we assign 100% probability. If the estimation begins from an unknown state, we create one tracker per possible system state and assign them equal probabilities. Assuming no prior knowledge on the mouse location, we set the mouse uncertainty area to cover the entire screen in each tracker during initialization.

The state and location estimation is an event-driven process. Based on the received user input events, we update the trackers on the list, create new trackers and delete existing ones from the list. For each mouse movement event, we update the mouse uncertainty area in each tracker. For every mouse click, we consider all possible outcomes of the click, including transitions to new states, as well as remaining in the same state. We create new *child trackers* with updated uncertainty areas, add the children to the list, and remove the *parent tracker* from the list (see Figure 4.6). When we observe a user event sequence that indicates interaction with a specific UI element, we update the probabilities of each tracker accordingly. We

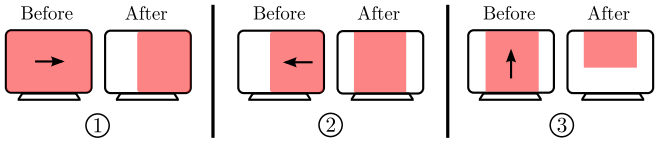


Figure 4.7: *Movement event handling. Movement can reduce the uncertainty area size (1) and (3) or change its location (2).*

explain these steps in detail below.

Movement event handling. When the mouse uncertainty area is the entire device screen, any mouse movement reduces the size of the uncertainty area. For example, if the user moves the mouse to the right, the area becomes smaller, as the mouse cursor can no longer reside in the leftmost part of the screen (Figure 4.7). If the mouse is moved to a direction where the uncertainty area border is not on the edge of the screen, the mouse movement does not reduce the size of the uncertainty area, but only causes its location to be updated. Any mouse movement towards a direction where the uncertainty area is on the border of the screen, reduces the size of the uncertainty area further. For each received mouse movement event, we update the uncertainty areas in all trackers.

Click event handling. When we observe a mouse click event, the estimation algorithm considers all possible outcomes for each tracker. The possible outcomes are determined by the current mouse uncertainty area (Figure 4.8). We create new child trackers for each possible outcome and update their mouse uncertainty areas as follows.

If the user interface remains in the same state, the updated mouse area for the child is the original area of the parent from which we remove the areas of the user input elements that cause transitions to other states. For each state transition, the mouse area is calculated as the intersection of the parent area and the area of the user input element that caused the transition. Once the updated mouse uncertainty areas are calculated for each child tracker, we remove the parent tracker from the list, and add the children to it. We repeat the same process for each state tracker on the list. We note that as a result of this process, the list may contain multiple trackers for the same state with different mouse uncertainty areas.

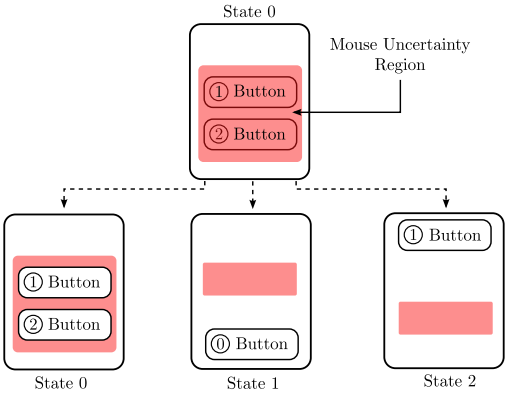


Figure 4.8: *Click event handling. We create state trackers for all possible click outcomes, including remaining in the same state and transitions to new states. A new uncertainty area is calculated for each tracker.*

The probability of a child tracker is calculated by multiplying the probability of its parent with a *transition probability*. We consider two options for assigning transition probabilities, as shown in Figure 4.9.

- *Equal transitions.* Our first option is to consider all possible state transitions equally likely. For example, if the mouse uncertainty area contains two buttons, each of them causing a separate state transition, and parts of the screen where a click does not cause a state transition, we assign each of them 1/3 probability.

- *Element area.* Our second option is to calculate the transition probabilities based on the surface of the user interface element covered by the mouse uncertainty area. For example, if the uncertainty area covers a larger area over one button than another, we assign it bigger transition probability.

The transition probabilities can be enhanced with *a priori probabilities* of UI element interactions. For example, based on prior experience on comparable user interfaces, the adversary can estimate that an OK button is pressed twice as likely as a cancel button in a given state.

Element detection. Finally, we identify user interaction with certain UI elements based on sequences of observed user input events. For example,

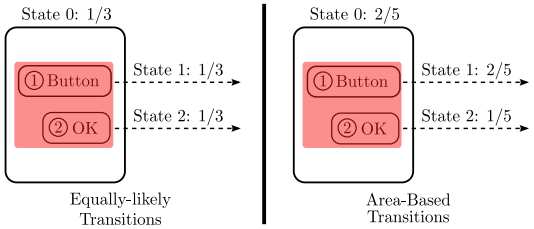


Figure 4.9: *Transition probabilities. On the left, all possible outcomes are considered equally like. On the right, we illustrate area-based transition probabilities (the larger the element, the more likely it was clicked).*

a mouse event sequence that begins with a button down event, followed by movement left or right that exceeds a given threshold, followed by a button up event is an indication of slider usage. Similarly, text input from the keyboard indicates likely interaction with an editable text field and a click indicates a likely interaction with a button.

When we observe such event sequences (slider movement, text input, button click), we update the probabilities of the possible trackers on the list. One possible approach would be to remove all trackers from the list where interaction with the identified element is not possible (e.g., a button click is not possible under the mouse uncertainty area). After such trackers would be removed from the list, we could increase the probabilities of the remaining ones equally. Such an approach could yield fast results, but also provide erroneous state estimations. If the user provides text input on a user interface state that does not contain editable text fields or if text highlighting is mistaken for slider movement, the algorithm would remove the correct state from the list. We adopt a safer approach where we consider trackers with the identified elements more likely and scale up their probabilities, and keep the remaining trackers and scale down their probabilities. The scaling factor is an adjustable parameter of this approach.

Target state detection. Our algorithm continues the state tracking process until two criteria are met. First, we have identified the target state with a probability that exceeds a threshold. After each click event and detected element we sum the probabilities for all trackers that represent the same state to check if any of them exceeds the threshold. Second, the mouse uncertainty area must be small enough to launch the attack. We combine the mouse uncertainty areas from all matching trackers and consider the

uncertainty area sufficiently small when its size is smaller than the size of the target elements or the confirmation element.

State estimation for touchscreen. Using a touchscreen instead of a mouse does not affect our algorithm. Typically touchscreens report click events in absolute coordinates, hence using a touchscreen corresponds to the case where the mouse location is known, but the starting state is not. Determining the possible transitions after a click is trivial, since there can be at most one intersection of a clicking point with the area of an element in a specific state. Text input can be observed from the virtual keyboard therefore the element detection works the same way as described previously.

4.4.3 User Interface Fingerprinting

In this section we describe UI fingerprinting that is applicable to adaptive UI attacks on general-purpose PC platforms. On a high level, our UI fingerprinting approach works as follows. The attack device keeps a history of all events observed in the last t minutes (in our experiments, $t = 5$min produced good results). For every observed mouse click event, the attack device takes the target application UI model, analyzes the event history and asks the following question: *"Is the user interacting with the critical UI?"*.

Listing 4.1: *Example of an e-banking UI fingerprinting model.*

```
el_account··········= Rect(...), T_REQ | T_ATTACK_EL, 0
el_amount·········· = Rect(...), T_REQ, 1
el_reference_number = Rect(...), T_REQ, 2
el_execute_date···· = Rect(...), T_REQ, 3
el_booking_text···· = Rect(...), T_OPT, 4
el_debit_check_box··= Rect(...), T_OPT, 5
el_total_check_box··= Rect(...), T_OPT, 6
el_next_button·····= Rect(...), T_REQ | T_CONFIRM_EL, 7

0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6
```

Similarly to our state tracking approach, we also require a UI model for fingerprinting. However, the UI model in this case is simpler, as no UI transitions are modeled. In Listing 4.1 we provide a real example of a UI model from one of the e-banking websites. The model consists of two parts. The first part specifies the elements, their relative positions, and their types (required or optional). The second part is the *tabbing order*, i.e., the order

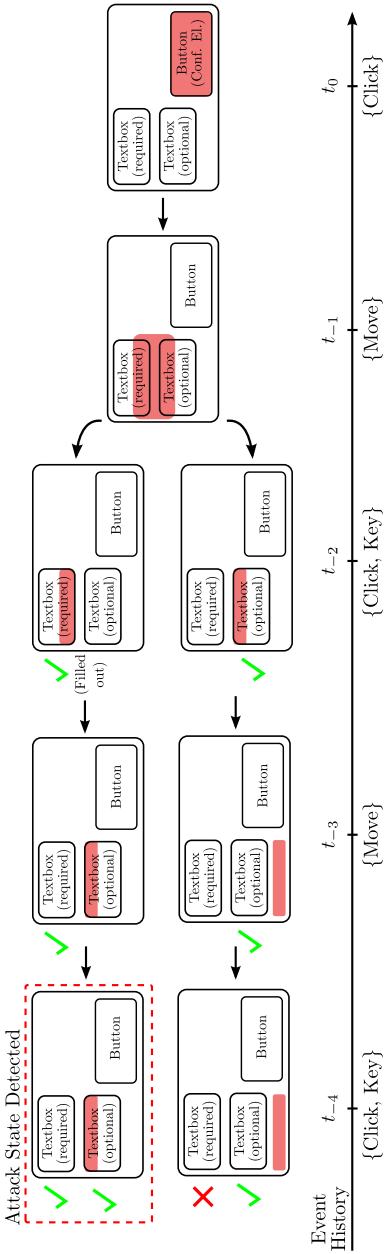


Figure 4.10: *Fingerprinting UI example. For every observed click, the attack device traverses the event history backwards in time, and checks if the user is currently interacting with the critical (target) UI.*

in which the elements are traversed in case a tab key is pressed. We note that specifying arbitrary UIs is simple using our notation.

In Figure 4.10, we illustrate our fingerprinting approach on a concrete example. The critical UI in this case is simple, and consists of three elements: two text-boxes and a button. We require one text-box to be filled out with keyboard input, while the other is optional. The latest event (at time t_0) the attack device observes is a mouse click. The attack device assumes the click was performed on the confirmation element, and traverses the event history backwards in time to check if the user was indeed interacting with the critical UI, according to the specified model.

The first encountered event is a mouse move, so the device moves the mouse uncertainty region accordingly. The following two events are a click and a key press, so the device creates two trackers (the uncertainty region was over two different elements), and shrinks the uncertainty region over the corresponding elements. The next event in the history is another move, followed by a click and a keyboard press. In the lower tracker, the click would have originated over no element, but in the upper tracker, both required text-boxes would be filled with text, at which point the attack device concludes that the user is interacting with the targeted UI state.

4.4.4 Attack Launch Techniques

Once the attack device has identified the attack state with sufficiently small uncertainty area, it is ready to launch the attack. In a simple approach, the adversary moves the mouse cursor over one of the attack elements, modifies its value, moves the mouse cursor over the confirmation button, and clicks it. The process is fast and the user has little chances of preventing the attack. However, the user is likely to notice such an attack. For example, if the doctor never clicked the confirm button, the doctor may be reluctant to implant the pacemaker into a patient. For this reason, we focus on more subtle attack launch techniques. Below we describe two such techniques and in Section 4.6 and Section 4.7 we evaluate their user detection.

Element-driven attack. The adversary first identifies that the user interacts with one of the target elements. This can be easily done when the mouse uncertainty area is smaller than the target element. Once the user has modified the value of the target element, the adversary waits a short period of time and during it tracks the mouse movement, then quickly moves the mouse cursor back to the target element, modifies its value, and returns the mouse cursor to its location. After that, the adversary lets the legitimate user confirm the safety-critical operation. The technique only

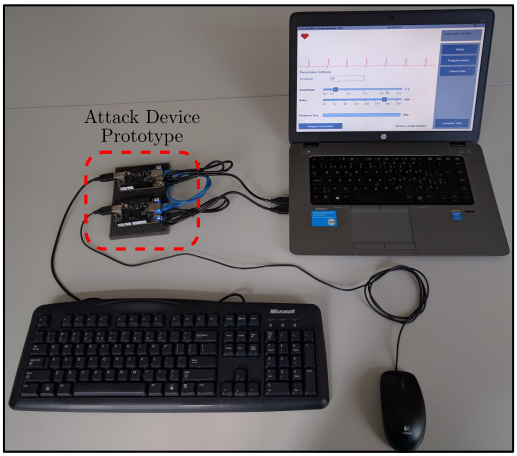


Figure 4.11: *Attack device prototype. It consists of two Beagle Bone Black boards that run the state estimator and attack launch implementation.*

requires little mouse movement, but the modified value remains visible to the user for a potentially long time, as the adversary does not know when the user will confirm the safety-critical operation.

Confirmation-driven attack. The adversary identifies that the system is on the attack state and lets the user to set the attack element values uninterrupted. When the user clicks the confirmation button, the attack activates. The adversary blocks the incoming click event, moves the mouse cursor over one of the attack elements, modifies its value, moves the mouse cursor back over the confirmation button, and then passes the click event to the target system. After that, the adversary changes the modified attack element back to its original value. In this technique, the mouse cursor may have to be moved more, but the modified attack element settings remain visible to the user only for a very short period of time.

4.5 Attack Device Protoype

We built a prototype of the attack device by implementing the entire attack system in C++ and deployed it on two BeagleBone Black boards (Figure 4.11). The two boards communicate over ethernet, because each board has only one set of USB ports and we evaluate an attack where the adversary controls both mouse and keyboard input. A custom attack device

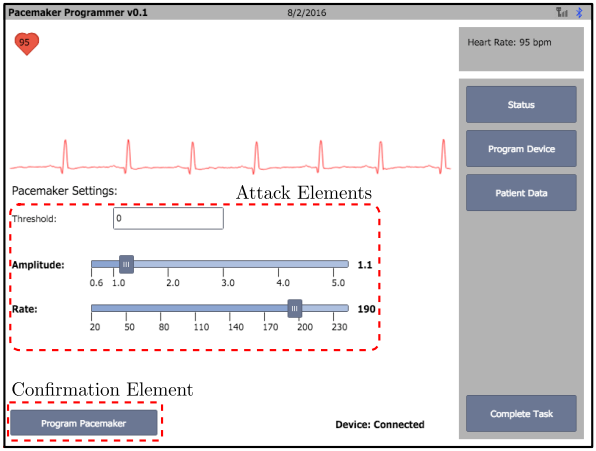


Figure 4.12: *Case study UI: custom cardiac implant programmer. The attack and the confirmation elements are highlighted.*

would consists of a single embedded device. The BeagleBone boards we use have processing power comparable to a modern low-end smartphone (1GHz CPU, 512MB RAM).

The boards are conveniently powered through USB, and no external power supplies are required. Each board intercepts one USB device (keyboard and mouse, respectively), and the two boards communicate through a short ethernet cable. We emphasize that the complete attack software is running on the boards themselves, and no remote communication with the attacker is either required or performed. We purposefully optimized the C++ code for execution speed.

4.6 Case Study: Pacemaker Programmer UI

To evaluate our state tracking based attack on terminals, we focus on a simulated pacemaker programmer user interface (Figure 4.12). We implemented the user interface based on the publicly available documentation of an existing cardiac implant programmer [32]. Such a programmer terminal is used by doctors to configure medical implant settings. For example, when a doctor prepares a pacemaker for implantation, the doctor configures its settings based on the heart condition of the receiving patient. The terminal can also be used to monitor the implant and potentially update its settings, and the user interface was designed for both mouse and keyboard use.

The model of this user interface consists of approximately ten states and contains three types of user input elements: buttons, text fields and sliders. All state transitions are triggered by button clicks. The attack elements are the user input elements that are used to configure the pacemaker settings. Threshold is set using a text field (keyboard), while amplitude and rate are set using slider elements (mouse), see Figure 4.12. All attack elements are on the same state. The model creation was a manual process that took a few hours. We load the UI model to the attack device prototype.

We implemented the user interface using the Haxe language that was compiled to the HTML5 (JavaScript) backend. The UI implementation serves two different purposes. First, we use it to demonstrate the attack and evaluate attack detection on-site. For this use, we run the user interface on a standard PC, instead of a terminal device (Figure 4.11). Second, we use the same UI to collect user traces and evaluate the detection of different attack variants online.

4.6.1 Trace Collection

To evaluate the tracking algorithm we collected user traces for the programmer user interface online.

Participant recruitment. We recruited 400 participants for trace collection using the crowd sourcing platform CrowdFlower. The platform enables the definition of typically small online jobs that human contributors complete in return of a small payment. We recruited participants globally and required them to be at least 18 years old. Each contributor was allowed to complete only one job for trace collection. On CrowdFlower platform our job had a title *"Program an implanted pacemaker"* and its description stated:

> "*We are evaluating the user interface of an experimental medical device. Your task is to configure a pacemaker device by interacting with the pacemaker programming software. Note that this is a test! The shown user interface is not connected to a real patient.*"

Task details. In each job, we asked the participants to fill in a short questionnaire that we used to collect demographics information. The questionnaire also included a test question, with a known answer, that we used to filter out participants that were clearly not attentive. After the questionnaire, the participants were shown more detailed task instructions and the pacemaker programmer user interface. The participants interacted with the user interface using a mouse and a keyboard on their browsers.

In the instructions, we asked the participant to find saved patient data that matches a given medical condition, copy that patient's pacemaker

Age	
18-29	41%
30-39	41%
40-49	13%
50-59	4%
60 or above	1%

Gender	
Male	74%
Female	26%
Education	
Primary school	2%
High school	32%
Bachelor	66%

Table 4.1: *User trace collection demographics.*

settings to the programming screen, and finally, to program the device by pressing the confirmation element. They remained visible to the participant while they interacted with the UI. We recorded all user input during the task, but no private information on study participants was collected.

Trace analysis. In total, 400 contributors completed the task and Table 4.1 provides their demographics. We divided the collected traces randomly into 200 training traces and 200 evaluation traces, and we analyzed all the training traces. The time required to complete the task varied greatly. The traces had 29 (±22) clicks on average, and 98% of the traces had at least ten clicks. We profiled each user interface state and calculated how often each button was pressed. We also analyzed conditional button press frequencies, i.e., how often a button was pressed given that the user transitioned to the current state from a given previous state. By analyzing the traces, we observed that approximately 7% of user input gestures were over wrong or non-existent elements. For example, users sometimes habitually clicked when the mouse cursor was not over a button element.

4.6.2 Estimation Accuracy

We ran our estimator implementation on all our evaluation traces. As our algorithm is event-based, after each click we measured (a) the size of the mouse uncertainty area, expressed as the percentage of the overall screen size, and (b) the probability that we correctly estimate the real state the user is currently in. Figure 4.13 shows our results.

We say that our algorithm correctly estimates the current state when it assigns the highest probability for the correct state among all states. As all our traces start from the same state, to evaluate the situation where the tracking begins from an unknown state, we cut the first 10% from all our evaluation traces. As tracking options, we used the element-area transition

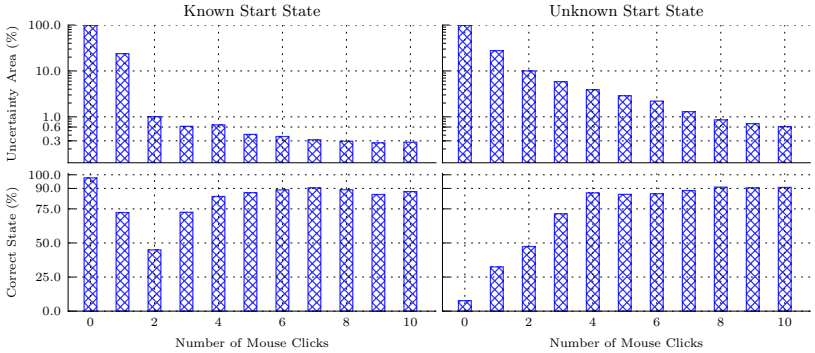


Figure 4.13: *State tracking accuracy. On the left, tracking from a known state. The uncertainty area reduces fast, and after ten clicks we estimate the correct state with high probability. On the right, tracking from an unknown start state. The uncertainty area reduces slower and we estimate the correct state with slightly lower probability.*

probabilities together with element detection (scaling parameter 0.95) and a priori probabilities that we obtained by profiling the training traces.

First, we discuss the case where the state tracking begins from a known start state (shown left in Figure 4.13). The uncertainty area is the full screen at first and the probability for estimating the correct state is 100% (known start state). As the estimation algorithm gathers more user input events, the uncertainty area size reduces quickly and already after three clicks the area is less than 1% of the screen size. The estimation probability decreases first, as the first click adds uncertainty to the tracking process, but after additional click events, the probability steadily increases, and after ten clicks the algorithm can estimate the correct state with 90% probability.

Next, we consider the scenario where the state tracking begins from an unknown target system state (shown right in Figure 4.13). In the beginning, the uncertainty area is the entire screen and the probability for the state estimate is low, as all states are equally likely. As the tracking algorithm gathers more user events, the uncertainty area reduces, but not as fast as in the case of known start state. The uncertainty area becomes less than 1% of the screen size after eight clicks. The probability for the correct state estimate increases and after ten clicks we can estimate the correct state

with approximately 90% probability. We do not report results past ten clicks, as many of our traces were shorter than that.

We conclude that in both cases we can identify the correct system state with high probability after observing only ten clicks, and the uncertainty area becomes very small (below 1%, which equals to a small 50×50 pixel rectangle). We note that in our user interface, the target and confirmation elements are significantly larger than 1% of screen size. If the user enters the attack state after ten clicks, we can launch the attack accurately.

In an older version of our implementation, we compared the performance of our different tracking options. We evaluated the two transition probability assignment schemes (equal transition and element area) and tested both schemes with and without element detection and a priori probabilities. We measured the probability that we estimate the correct state after ten clicks and we noticed that both transitions option performed comparably. Element detection gave a major detection accuracy improvement. A priori probabilities did not improve accuracy significantly.

4.6.3 Attack Launch Success Rate

To evaluate if our system successfully detects the correct time to launch the attack, we ran through our algorithm all the user traces we recorded from our user study, in an offline manner. In 83% of traces, our system correctly identified the attack launch time, namely right after the user programs the pacemaker. In 16% of the traces, our system did not identify a suitable attack time and, as a result, no attack would be launched. Only in 1% of the traces our system launched the attack at the wrong time. We conclude that our system correctly identifies the attack launch time in most cases.

4.6.4 Estimation Overhead

To analyze how fast our state and location estimation algorithm runs, we measured the runtime overhead of processing each user input event from the collected 200 evaluation traces. The estimation algorithm was run on the two BeagleBone boards (1GHz, 512MB) with element tracking enabled, using equal transition probabilities.

Both mouse movement and keyboard events require little computation. The processing overhead per event is very small (below 0.5ms) and such events can be easily processed in real-time. Mouse click events require more computation, as those cause generation of new trackers, and the processing delay is relative to the number of state trackers that the algorithm maintains. Figure 4.14 shows the average processing delay for mouse clicks from

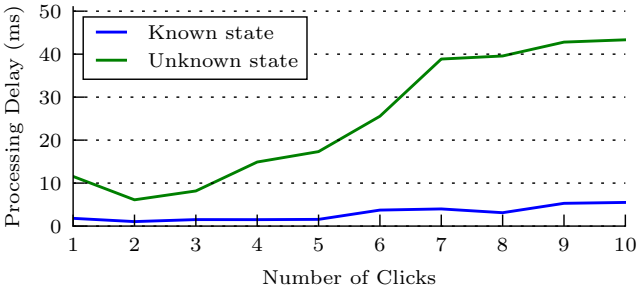


Figure 4.14: *State tracking overhead on BeagleBone boards. The tracking overhead per user input event increases over time as the state tracking algorithm accumulates more trackers, but is overall low.*

our evaluation traces. When we start tracking from a known state, the overhead increases slightly over time, but remains under 7ms per event. When we start tracking from an unknown state, the algorithm accumulates significantly more trackers, and thus the processing overhead increases faster. After ten clicks, processing a single input event takes approximately 43ms on our test platform. From the analyzed traces we observed that the interval between consecutive clicks is typically in the order of seconds, which gives the attack device ample time to process incoming click events.

We conclude that our implementation is fast. Mouse movement and keyboard events are processed in real-time and the processing overhead for mouse clicks is significantly smaller than the typical interval between clicks. The target UI remains responsive, with no observable "processing lag" that would indicate to the user that an attack is taking place.

4.6.5 Online Attack Detection User Study

To evaluate how many users would detect our attacks, we conducted two user studies. Here we describe the first, large-scale online study.

Recruitment and procedure. We created a new job on the same crowd sourcing platform with similar description and recruited 1200 new study participants. We divided the participants into 12 equally large attack groups of 100 participants each. We tested two element-driven attack variants: one where we modify a text input element and another we modify a slider input element. We also tested two confirmation-driven attack variants: one

Age	
18-29	42%
30-39	39%
40-49	13%
50-59	4%
60 or above	2%

Gender	
Male	70%
Female	30%
Education	
Primary school	2%
High school	30%
Bachelor	68%

Table 4.2: *Online attack detection study demographics.*

with text and another with slider input. For each four attack variant we tested three separate speeds of the attack.

For all participants, we provided the same task description as before, but depending on the group, we launched an attack during the task. Once the task was over, we asked the participants: *"Do you think you programmed the pacemaker correctly?"* with yes/no answer options. We also asked the participants to give freeform feedback on the task.

If participants noticed the UI manipulation, they had three possible ways to act on it. First, the participants were able to program the pacemaker again with the correct values. Second, the participants could report that the device was not programmed correctly in the post-test question. Third, the participants could write to the freeform feedback that they noticed something suspicious in the application user interface.

Study results. In total, 987 participants completed the task and we report their demographics in Table 4.2. We consider that the attack succeeded when the participant did none of the above mentioned three actions, and the results are shown in Table 4.3. The success rate for the element-driven text attacks was 37-50% and for the element-driven slider attacks 6-12%, depending on the speed of the attack. The success rate for the confirmation-driven text attacks was 93-96% and for the confirmation-driven slider attacks 90-95%.

All the tested confirmation-driven attacks had high success rates (over 90%). In the element-driven attacks the UI manipulation remains visible longer, and this is a possible explanation why the attacks do not succeed equally well. We conclude that confirmation-driven attacks are a better strategy for the adversary and focus the rest of our analysis on those.

We compared the success rates of text and slider manipulation on confirmation-driven attacks, but found no statistically significant difference ($\chi^2(1, N = 484) = 0.12), p = 0.73$). This implies that both variants are

Attack group	Attack succeeded	Task completed
1. Element, text, 5 ms	50%	84
2. Element, text, 62 ms	37%	84
3. Element, text, 125 ms	48%	86
4. Element, slider, 5 ms	12%	80
5. Element, slider, 62 ms	9%	83
6. Element, slider, 125 ms	6%	86
7. Confirmation, text, 10 ms	93%	81
8. Confirmation, text, 125 ms	96%	79
9. Confirmation, text, 250 ms	93%	78
10. Confirmation, slider, 10 ms	95%	85
11. Confirmation, slider, 125 ms	90%	82
12. Confirmation, slider, 250 ms	95%	79
Total		**987**

Table 4.3: *Attack detection study results. For each attack group we report the the percentage of users against which the attack succeeded, and the number of participants that completed the task.*

equally effective. We also compared the success rates of different attacks speeds on confirmation-driven text attacks ($\chi^2(2, N = 238) = 0.42), p = 0.81$) and slider attacks ($\chi^2(2, N = 246) = 2.20), p = 0.33$), but found no significant difference. This implies that the adversary has at least a few hundred milliseconds of time to perform the user interface manipulation without sacrificing its success rate.

4.6.6 Discussion

We analyzed the freeform text responses, and none of the users associated the observed UI changes to a malicious attack. Only two users commented on the changing values of UI elements and both attributed the changes to a software glitch. One user noted *"Possible bug when working with sliders. Threshold value changed from 88 back to 80, had to correct"*. This result shows that users are habituated to software errors in user interfaces.

Out of the 987 study participants only 21 answered negatively to the question *"Do you think you programmed the pacemaker correctly?"*. A possible explanation is that users misinterpreted the results of positive UI feedback. To reduce chances of errors, it is common for safety-critical systems to have strong positive feedback mechanisms (e.g., clearly visible

user action notifications). Even though an attack was performed, the users could have been fooled into believing that nothing out of the ordinary happened by the reassuring nature of our "*Device Programmed*" notification. A user remarked: *"I was told at the end that the pacemaker was programmed, so I assume I did it correctly"*.

4.6.7 On-Site Attack Detection User Study

Our online study participants were not domain experts, i.e., users that would commonly operate a pacemaker programmer. To understand how domain experts react to our UI attacks, we performed a preliminary on-site user study on two medical professionals (one male doctor and one female nurse). The doctor was 33 and the nurse 24 years old. The doctor has a specialization in internal medicine, and the nurse works in the field of anesthesiology. Both study participants regularly work with medical devices that have similar UI complexity to our simulated pacemaker programmer.

The participants had the same task as in our online study: to program the pacemaker with patient-specific values. Our on-site study setup is similar to Figure 4.11. The pacemaker programmer application is running on a laptop, to which a USB keyboard and mouse are connected. The attack device is installed, but was hidden from sight. Both participants failed to detect our attack, and were under the impression they programmed the pacemaker correctly.

The results of this follow-up and on-site user study give further confidence on the previous results of the large-scale online study, namely that our attacks are difficult to detect.

4.7 Case Study: Online Banking UI

In this section we describe our experiments on e-banking user interface. We first explain our trace collection and analysis, followed by our user study.

4.7.1 Trace Collection and Analysis

We installed keyboard and mouse logging software on the author's laptops. The programs collected every mouse and keyboard event, and stored them in a heavily filtered manner (e.g., we did not log which key was pressed, but only if the "tab" key was pressed or some "other" key was pressed). We collected over 72 hours worth of regular mouse and keyboard usage patterns. During those 72 hours, the authors never visited real e-banking websites. We then fed all traces through our attack code. The code had no

false positives, i.e., the code did not erroneously detect that an e-banking website was used, when in reality it was not.

To analyze how fast our fingerprinting algorithm runs, we measured the runtime overhead of processing each user input event from the collected traces. The estimation algorithm was also run on the two BeagleBone boards. Both mouse movement and keyboard events require little computation. The processing overhead per event is very small (below 0.5ms) and such events can be easily processed in real-time. Mouse click events require more computation, as it is at this point where the algorithm determines whether the user is interacting with the critical UI. Since we limited the history to a 5 minute interval, the processing delay remains fairly constant, averaging approximately 49ms and for 96% of the clicks the delay is less than 100ms on our test platform.

4.7.2 On-Site Attack Detection User Study

To evaluate how successful our attack is in fingerprinting UIs and compromising the integrity of e-banking payments, we performed a separate on-site user study, similar to the previous on-site attack detection study. We created partial local replicas of three major e-banking websites (we only copied the payment parts of the sites). The replicas were nearly the same as their online counterparts, with minor differences inadvertently introduced through the replication process.

Recruitment. We recruited 20 domain experts. To participate in our study, each participant was required to be a regular e-banking user of one of the three banks we replicated the websites of, and use either Chrome of Firefox during e-banking sessions.

Procedure. Each participant was presented with a sheet of paper, containing the following instructions steps.

> *"(1) Open the browser you usually use for e-banking. (2) Click on your e-banking site link, located in the browser bookmarks. (3) Imagine you already performed the login procedure, as the replica website requires no login. (4) Navigate to the payment site, and (5) make a payment to the account provided on the study sheet. (6) To complete the task, close the browser."*

As in our previous user study, the attack device was already installed to the laptop and was hidden from sight. First step of our attack was

Attack results	
Attack successful, not noticed	80%
Attack successful, noticed	10%
Attack failed, not noticed	5%
Attack failed, noticed	5%

Table 4.4: *Online banking UI user study results.*

to automatically detect that the user is interacting with a critical UI, i.e., that payment information is filled in, and which of the three banks was being used. The second step of the attack was to detect when the "Confirm payment" button was clicked. The attack device then inserted mouse and keyboard input that changed the payment field containing the amount of money transmitted, and injected a small javascript snipplet through the URL bar, that masked the changed value in the upcoming "Payment confirmation" screen. The whole attack was done in approximately 0.5 seconds. A video which demonstrates our attack is available online[3].

After completing the user study, we presented each participant with an exit questionnaire, consisting of two questions: *"1. Was the payment experience comparable to your regular e-banking experience?"* and *"2. What do you think the user study was about?"*

Results. We present the results of our attack in Table 4.4. Our attack successfully detected the precise point in time when the users were interacting with the critical UI (making a payment) in 90% of the cases. Our attack failed in only 10% of the cases (two users). In both of these cases the attack state detection succeeded, but the attack input event injection failed in both cases, due to implementation flaws.

Over 90% of participants positively answered to the first question, noting that it was similar to their regular e-banking experience. Out of those, some users noted that the UI looked slightly different, which was due the imperfections introduced by our replication process. Only 10% noted that the experience was not the same, due to the missing second-factor authentication step. Out of 20 participants, 30% had no idea about the true nature of the user study. Another 30% suspected that some form of attack was performed (phishing, key-logging, removal of second-factor authentication), while another 30% thought the study was a usability test.

[3] `https://goo.gl/kdkRDC`

Only two users detected our attack, and correctly guessed the true nature of the study. We conclude that our attack was stealthy to most users.

4.7.3 Discussion

We did not perform any security priming in our user study, however we acknowledge that the role-playing bias of study participant not using their real e-banking could be present. Users might have been less careful, because they knew that their own money is not at risk. At the same time, our study setup introduces another bias. Since the study was performed under our supervision, some study participants may have been more alert than if they would have done the online payment on their own.

We tested our attack on various browsers, various e-banking sites and various screen resolutions as well as browser window locations, and we conclude that our attacks can successfully perform UI fingerprinting and e-banking session hijacking, with no false positives, and very low false negatives. Furthermore, we showed that our attacks were not detected by the majority of users.

The user study was performed on a custom laptop that we provided, that was disconnected from the internet. The e-banking website replicas required no logins of any kind, and at no point in time did we require the study participants to disclose any kind of private information, such as their their e-banking credentials. No real money was exchanged during our study, as the performed e-banking transaction were purely fictional. We de-briefed each study participant at the end of the user study, where we described what the user study was about and we reassured each participant that no real transaction was performed, and that their e-banking credentials were not compromised in any manner.

4.8 Countermeasures

In this section we analyze possible countermeasures and their limitations regarding the attacks presented in this chapter.

Trusted input devices. One way to address our attacks is to mandate usage of trusted input devices. We call a user input device *trusted*, when it securely shares a key with the target system. For example, USB input devices communicate using polling. The host sends periodic requests and the input device sends responses that report a possibly occurred user event. With a shared key all request and responses can be encrypted and authenticated which prevents the adversary from observing and injecting events. If the

responses also include freshness, such as a nonce, the adversary cannot replay events either.

However, secure deployment of trusted input devices is challenging. Assuming that the target system and the input device have a certified key, and the two devices run a mutually authenticated key agreement protocol at connection establishment. If the certified input device is temporarily unavailable (e.g., lost or broken), the safety-critical terminal cannot be operated with another, non-certified device. For example, doctors need be able to operate medical terminals at all times. Additionally, the adversary can purchase a certified input device, extract a key from it, and install it to the attack device. Standard approaches to address compromised keys, such as online revocation checks, are ill-suited to this setting, as many safety-critical embedded terminals are disconnected from the Internet to limit their attack surface.

Increased user feedback. The user interface can provide visual feedback on each change on attack elements [90]. For example, the user interface can draw a thick border around a recently edited element and keep it visible for some amount of time. In a confirmation-driven attack, the user would see the border, but as the adversary changes the attack element value back to the original, the content of the user interface element would appear as expected. Noticing such attacks may not be easy for the user.

Change rate limiting. The user interface could limit rate at which the values of the user interface elements can be changed. However, our study results show that the majority of users do not notice even relatively slow UI manipulations that take 250ms. Finding a rate limit that efficiently prevents user interface manipulation attacks, but does not prevent legitimate user interactions can be challenging.

Randomized user interfaces. Another way to address our attacks is to randomize parts of the safety-critical system user interface. Both our state tracking algorithm and the attack launch techniques assume a static model of the target system user interface. If the user input elements change their location for every execution, the system state tracking becomes significantly harder. Also attack launch can be complicated by using randomized element locations. Randomized user interfaces have been proposed for smartphone screen lock to prevent shoulder surfing and smudge attacks [174, 185]. For example, the Intel IPT technology randomizes PIN input to prevent malware from stealing it [91].

Attack type	CAPTCHA placement	
	Confirmation	Element mod.
confirmation-driven	attack possible	attack prevented
element-driven	attack possible	visibility increased

Table 4.5: *Placement options for human user tests.*

While UI randomization can complicate, or even prevent our attacks, it also increases the chances of human error. In contrast to smartphone screen lock, on safety-critical terminals an increased error rate is typically not acceptable. For example, medical device evaluations consider lack of UI consistency a critical safety violation [80]. Randomization can increase attack resistance, and thus improve safety, but at the same time increase human errors, and thus decrease safety. Finding the optimum is an interesting direction for further research, but outside the scope of this chapter.

Human user tests. Passwords are often used to authenticate that the correct user is interacting with a computing system. Passwords do not protect against our attacks, because the adversary can learn any entered passwords. CAPTCHAs are a common technique to verify that the user input originates from a human. In our scenario, the attack device cannot solve a CAPTCHA, as it cannot read from the screen, and observing the user to solve one test does not help in future tests.

The terminal user interface can require that the user must solve a human user test to confirm the safety-critical operation. This approach does not prevent element-driven attacks. Once the adversary has detected interaction with the attack elements, it can wait, modify their values, and after that let the user to complete the test in order to confirm the operation. Also confirmation-driven attacks remain possible with this approach. The user can also be asked to solve a test to be allowed to modify the attack elements. This prevents confirmation-driven attacks. When the user chooses to confirm the safety-critical operation, the adversary cannot return to the attack elements and modify their values without user involvement. For element-driven attacks the adversary has to adjust his attack strategy. The adversary must perform the modification when the user interacts with one of the attack elements (and not shortly after it). Table 4.5 summarizes these options. Human user tests can improve attack resistance, but forcing the user to solve such a test for every modification of a safety-critical UI

element is not be acceptable in many systems we consider.

Continuous user authentication. While traditional user authentication systems require the user to log in once, continuous authentication systems monitor user input over a period of time to detect if the usage deviates from a previously recorded user profile. Many such systems track mouse velocity, acceleration and movement direction [38, 165], together with click events [38, 165], angle-based curvature metrics and click-pause measurements [211]. Typically these systems collect user input events for a fixed period of time and then analyze the input to detect unauthorized usage.

The proposed systems that demonstrate low false rejection rates, typically require a significant number of consecutive impostor actions (e.g., 20 consecutive mouse clicks [211] or 70 consecutive mouse actions [135]). Even when tailored for higher false rejection rates, the systems need to observe the impostor for significant amount of time (e.g., 12 consecutive seconds [164]). Our attacks require only brief mouse movement and one or few clicks, and the attacks can be performed well under a second. Our state estimation works fully passively. Thus, the current continuous authentication systems are not directly applicable to detection of our attacks.

Summary. We conclude that all the reviewed countermeasures have some limitations. Finding better protective measures that are both effective and practical to deploy remains an open problem.

4.9 Discussion

In this chapter, we presented a novel adaptive approach to attack systems through input integrity violation under uncertainty about the target system state. In this section we discuss the applicability of the proposed approach to other scenarios and directions for future work.

User interface complexity. We experimented our attacks on a terminal user interface that consists of approximately ten states. We consider this typical UI complexity for embedded dedicated-purpose terminals. Evaluating our attack on more complex UIs would be an interesting direction for further work. Another improvement of our current work would be to extend our probabilistic tracking algorithms to handle non-deterministic UIs. We also experimented on real (and replica) online banking websites. These user interfaces represent typical complexity of many online services. While evaluation of all possible user interfaces is infeasible, we believe that

these examples capture different types of security-critical user interfaces.

System output. In our attack scenario, the adversary has only access to the user input channel. An attack device that is attached to an interface that connects a touchscreen to the terminal mainboard is an example of a scenario where the adversary may be able to access the system output channel as well. Video interfaces can have high bandwidths and running image recognition algorithms on a small embedded device may be challenging. In many scenarios, the adversary might be left with some uncertainty about the target system state, and our probabilistic tracking techniques can help launch a more accurate attack. In this chapter, we focused on tracking mouse and keyboard events, but many terminals are operated through touchscreens. State tracking based on touchscreens is easier compared to mouse events, as such events have no location uncertainty.

User presence. We tailor our attack for the case where the legitimate user is operating the device. The presence of the legitimate user both helps and complicates our attack. Observing specific input events (e.g., mouse clicks that presumably take place over buttons) help the adversary to determine the current user interface state. At the same time, the adversary must inject the attack events in a subtle manner to avoid user detection. If the attack is performed without the presence of the user (e.g., when the system is idle), a different strategy is needed for state estimation. Exploring such state estimation strategies is another interesting direction for future work.

Attacks in the wild. While the proposed attack approach is unconventional and the types of attacks described in this chapter have not been reported before, we believe that the attack scenario is realistic. In fact, similar attacks may already be taking place in the wild. For example, the NSA cottonmouth project[4] is a malicious USB connector that can both inject and observe user input. Such and similar devices are ideally suited to perform our attacks.

4.10 Related Work

In Chapter 2, we reviewed related work in the domain of user interface security, and in this section we compare our work to key areas in more detail.

USB attacks. Key loggers are small devices that the adversary can attach between a keyboard and the target system. The key logger records user

[4] `https://nsa.gov1.info/dni/nsa-ant-catalog/usb/index.html`

input and the adversary collects the device later to learn any entered user secrets such as passwords. Such attacks are limited to passive information leakage, while our approach enables active runtime attacks with severe safety or security implications.

A malicious user input device, or a smartphone that impersonates one [187], can attack PC platforms by executing pre-programmed attack sequences [40, 50, 123]. For example, a malicious keyboard can issue dedicated key sequence to open a terminal and execute malicious system commands. The input device might also be able to copy malicious code to the target system. Such attacks are typically not possible on hardened embedded terminals where the user cannot escape the application UI, and installation and execution of unsigned code is prevented.

USB firewalls. In recent research, USB firewall architectures have been proposed [19,179,180]. Similar to network firewalls, these architectures include packet filtering functionality (e.g., in the OS kernel). These firewalls can prevent a USB peripheral of one class masquerading as an instance of another class (e.g., mass storage device masquerades as keyboard). Such protective measures do not prevent our attacks, where all injected USB packets match the device class of the benign peripheral.

USB fingerprinting. Researchers have demonstrated fingerprinting of PCs based on their USB communication timing patterns [28]. Similar approach could be applied to fingerprint USB input devices. The processing delays that our attack incur are so small that users cannot observe them, but it remains an open question if timing-based fingerprinting could be used to detect the attack.

Terminal protection. Software-based attestation is a technique that allows a host platform to verify the software configuration of a connected peripheral [106]. Such attestation would address the variant of our attack, where the adversary replaces a benign user input device with a malicious one, but not the variant where the attack device sits between the benign peripheral and the terminal. Power analysis can be used to identify unknown (malicious) software processes running on embedded terminals, such as medical devices [46]. Such approaches would not detect our attack where no malicious code is running on the embedded terminal. Our concrete attack device prototype is susceptible to such power-analysis as it draws power from the host USB connection. However, the attack device could

easily be designed to include on-board battery power.

User deception attacks. In systems where multiple applications or websites share the same display, the user can be tricked to interact with false UI elements. For example, a malicious website may be able to draw an overlay over a button that causes the user click the button unintentionally. Such attacks are called clickjacking [90] or UI redressing [139]. In our attack scenario, the adversary can only modify and injects user events.

4.11 Conclusion

In this chapter, we have presented a new way to attack security-critical user interfaces. In the attack, the adversary installs an attack device between a user input device and the target system, and the attack is launched when the authorized user is performing a security-critical operation, by modifying or injecting new user input events. Our approach is easy to deploy on the location, invisible to traditional malware detection, difficult for the user to notice, and surprisingly robust to noise. Many of the attack variants we tested had success rate over 90%. We analyzed several countermeasures and noticed that all of them have limitations. We conclude that our attack presents a serious threat to many safety-critical terminals and general-purpose PC applications.

Attack limitations. Our attack approach is, by nature, a targeted attack. An attacker therefore needs to tailor it for the targeted system (or application) beforehand, by creating UI models or fingerprinting attack states.

Part II

Countermeasures

Introduction

In the first part of this thesis, our goal was to better understand the attack surface in the context of modern user interfaces. In the second part, we focus on countermeasures for a widespread and damaging kind of UI attacks; namely mobile application visual impersonation attacks, such as spoofing or phishing.

Applying existing malware detection approaches to phishing applications is challenging. There is a large body of work in the general domain of malware detection, and the underlying assumption is that malware behaves fundamentally differently from benign applications in a way that enables detection. However, compared to regular mobile malware, spoofing applications (e.g., phishing apps) can be considerably stealthier, as such malware does not necessarily perform any suspicious actions, other than drawing on the device screen.

A common approach is to extract similarity metrics by analyzing datasets of known malware. However, such approaches have drawbacks: (1) such malware datasets may not always be available, and (2) the extracted similarity features may not necessarily capture what users consider as similar. Furthermore, existing approaches are often susceptible to simple obfuscation attacks, where an attacker slightly modifies the malicious application in order to produce different fingerprints, and thereby evade detection. To detect spoofing apps in a manner that is accurate and resilient to obfuscation, a new approach is needed.

We observe that mobile application user interfaces are significantly simpler and visually cleaner when compared to common desktop user interfaces (Figure 4.15). We therefore take a conceptually different approach, and we use metrics based on *visual similarity*, through screenshot analysis and comparison.

Visual similarity offers new possibilities, as well as new challenges. In the following two chapters, we propose two systems for detecting visual impersonation attack. In Chapter 5, we propose a detection system that runs on the user's device, and in Chapter 6 we present an alternative (and complementary) system that runs on the marketplace and can detect malware prior to infecting the user's devices.

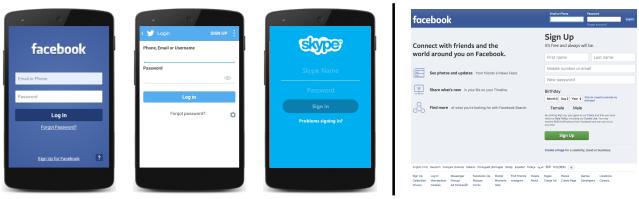


Figure 4.15: *Complexity of user interfaces. (Left) Mobile login screens commonly exhibit a clean design, with few user interface elements present. (Right) Desktop login screens are significantly more complex.*

Chapter 5

On-device Spoofing Detection

5.1 Introduction

Mobile application spoofing is an attack where a malicious mobile application mimics the visual appearance of another one. The goal of the adversary is to trick the user into believing that she is interacting with a genuine application while she interacts with one controlled by the adversary and, if such an attack is successful, the integrity of what the user sees (system output) as well as the confidentiality of what the user inputs into the system can be violated by the adversary. This includes login credentials, personal details that users typically provide to applications, as well as the decisions that they make based on the information provided by the applications.

A common example of mobile application spoofing is a phishing attack where the adversary tricks the users into revealing their password, or similar login credentials, to a malicious application that resembles the legitimate app. Several mobile application phishing attacks have been seen in the wild [118, 171, 203]. For example, a recent mobile banking spoofing application infected 350,000 Android devices and caused significant financial losses [69]. More sophisticated attack vectors are described in recent research [31, 42, 64, 197].

The problem of spoofing has been studied extensively in the context of phishing websites [2, 3, 56, 88, 93]. Web applications run in browsers that provide visual cues, such as URL bars, SSL lock icons and security

skins [55], that can help the user to authenticate the currently displayed website. Similar application identification cues are not available on modern mobile platforms, where a running application commonly controls the whole visible screen. The user can see a familiar user interface, but the interface could be drawn by a malicious spoofing application — the user is unable to authenticate the contents of the screen.

Security indicators for smartphone platforms have been proposed [61, 162], but their effectiveness relies on user alertness and they typically require either hardware modifications to the phone or a part of the screen to be made unavailable to the apps. Application-specific personalized indicators [121, 197] require no platform changes, but increase the application setup effort. Static code analysis can detect API call sequences that enable certain spoofing attacks [31]. However, code analysis is limited to known attack vectors and many spoofing attacks do not require any specific API calls, as they only draw on the screen.

We propose a novel spoofing detection approach that is tailored to the protection of mobile app *login screens* using visual similarity. Our system periodically grabs screenshots on the user's device and extracts visual features from them, with respect to *reference values* — the login screens of legitimate apps (on the same device) that our system protects. If a screenshot demonstrates high similarity to one of the reference values, we label the currently running app potentially malicious, and report it to the platform provider or warn the user. As our system examines screenshots, it is agnostic to the spoofing screen implementation, in contrast to approaches that examine screen similarity through code analysis. While straight-forward approaches based on visual similarity can detect simple cases of spoofing, where the attacker creates a perfect copy of the target app, or introduces other minor changes (e.g., changes the background color), our system can detect also more sophisticated spoofing.

In order to label spoofing apps accurately, our system needs to understand what kind of attacks are successful in reality, i.e., how much and what kind of visual similarity the two compared applications should have, so that the user would mistake the spoofing app as the legitimate one and fall for the attack. We capture this notion as a novel similarity metric called *deception rate*. For example, when deception rate is 20%, one fifth of the users are estimated to consider the spoofing app genuine and enter their login credentials into it. Deception rate is a conceptually different similarity metric from the ones previously proposed for similarity analysis of phishing websites. These works extract structural [12, 111, 154, 205, 208] as well as visual [43, 72, 125] similarity features and combine them into a similarity

score that alone is not expressive, but enables comparison to known attack samples [111, 129]. While existing metrics essentially tell how similar the spoofing app is to one of the known attacks, our metric determines how likely the attack is to succeed. Deception rate can be seen as a risk measure and we consider it a powerful new way to address spoofing attacks, especially in cases where a large dataset of known attacks is not available.

Our system requires a good understanding of how users perceive and react to changes within mobile app user interfaces. Change perception has been studied extensively in general [131, 151, 168], but not in the context of mobile apps. We conducted a large-scale online study on mobile app similarity perception. We used a crowd sourcing platform to carry out a series of online surveys where approximately 5,400 study participants evaluated more than 34,000 spoofing screenshot samples. These samples included modified versions of Facebook, Skype and Twitter login screens where we changed visual features such as the color or the logo. For most of the experimented visual modifications we noticed a systematic user behavior: the more a visual property is changed, the less likely the users are to consider the app genuine.

We used the results of our user study to train our system using common supervised learning techniques. We also developed novel visual feature extraction and matching techniques. Our system shows robust screenshot processing and good deception rate accuracy (6–13% error margin), i.e., our system can precisely determine when an application is so similar to one of the protected login screens that the user is in risk of falling for spoofing. No previous visual similarity scheme gives the same security property.

Additionally, we describe a novel collaborative detection model where multiple devices take part in screenshot extraction. We show that runtime detection is effective with very little system overhead (e.g., 1%). Our results can also be useful to other spoofing detection systems, as they give insight into how users perceive visual change.

5.2 Problem Statement

In mobile application spoofing, the goal of the adversary is to either violate the integrity of the information displayed to the user or the confidentiality of the user input. Application phishing is an example of a spoofing attack where the goal of the adversary is to steal confidential user data. The adversary tricks the user into disclosing her login credentials to a malicious app with a login screen resembling the legitimate one. A malicious stock market app that resembles a legitimate one, but shows fake market values,

is an example of an attack where the adversary violates the integrity of the visual information displayed to the user and affects the user's future stock market decisions. Below we review different ways of implementing application spoofing attacks.

The simplest way to implement a spoofing attack is a repackaged or otherwise cloned application. To the user, the application appears identical to the target application, except for subtle visual cues such as a different developer name. Application repackaging has become a prevalent problem in the Android ecosystem, and the majority of Android malware is distributed using repackaging [41, 213].

In a more sophisticated variant of mobile application spoofing, the malicious app masquerades as a legitimate application, such as a game. The user starts the game and the malicious app continues running in the background from where it monitors the system state, such as the list of currently running applications. When the user starts the target application, the malicious application activates itself on the foreground and shows a spoofing screen that is similar, or exactly the same, to the one of the target app. On Android, background activation is possible with commonly used permissions and system APIs [31, 64] and such attacks are difficult for the user to notice. While API call sequences that enable background attacks can be detected using code analysis [31], automated detection is complicated by the fact that the same APIs are frequently used by benign apps.

A malicious application can also present a button to share information via another app. Instead of forwarding the user to the suggested target app, the button triggers a spoofing screen within the same, malicious application [64]. Fake forwarding requires no specific permissions or API calls which makes such attack vectors difficult to discover using code analysis. Further spoofing attack vectors are discussed in related work [31].

Mobile application spoofing attacks are a recent mobile malware type and a large corpus of known spoofing apps is not yet available. However, serious attacks have already taken place. The Svpeng malware infected 350,000 Android devices and caused financial loss worth of nearly one million USD [69]. The malware presents a spoofed credit card entry dialog when the user starts the Google Play application and monitors startup of targeted mobile banking applications to mount spoofing attacks on their login screens. As spoofing detection using traditional code analysis techniques has inherent limitations and many spoofing attacks are virtually impossible for the users to notice, the exact extent of the problem remains largely unknown. Due to the already seen serious attacks, we believe it is useful to seek novel ways to address the problem of mobile app spoofing.

The problem of mobile application spoofing has many similarities to the one of web phishing. The majority of the existing web phishing detection schemes [12, 111, 154, 205, 208] train a detection system using a large dataset of known phishing websites. As a similar dataset is not available for mobile apps, these approaches are not directly applicable to mobile app spoofing detection. We also argue that the specific nature of mobile applications benefits from a customized approach, and in the next section, we introduce a novel detection approach that is tailored to mobile app login screens. The focus of this work is on mobile app spoofing and web phishing is explicitly out of scope.

5.3 Our Approach

In this section, we first describe the rationale behind our approach and introduce deception rate as a similarity metric. We then describe how this approach is instantiated into a case study on login screen spoofing detection. Finally, we describe our attacker model.

5.3.1 Visual Similarity and Deception Rate

The problem of application spoofing can be approached in multiple ways. Code analysis has been proposed to detect API call sequences that enable spoofing attacks [31]. However, code analysis is limited to known attack vectors and cannot address spoofing attacks that do not require specific API calls (e.g., fake forwarding). Another approach is to analyze the application code or website DOM trees to identify apps with *structural* user interface similarity [12, 111, 154, 205, 208]. A limitation of this approach is that the adversary can complicate code analysis, e.g., by constructing the user interface pixel by pixel. Third, the mobile platform can be enhanced with security indicators [61, 162]. However, indicator verification imposes a cognitive load on the user and their deployment typically requires either part of the screen to be made unavailable to the applications or hardware modifications to the device. Application-specific personalized indicators [121, 197] can be deployed without platform changes, but their configuration increases user effort during app setup.

In this chapter, we focus on a different approach and study the detection of spoofing attacks based on their *visual similarity*. Previously, visual similarity analysis has been proposed for detection of phishing websites [72, 194, 205]. Designing an effective spoofing detection system based on visual similarity analysis is not an easy task, and we illustrate the challenges by providing two straw-man solutions.

Figure 5.1: *Spoofing application example. The legitimate Netflix app and the Android.Fakeneflic malware [176]. The spoofed user interface includes subtle visual modifications.*

The first straw-man solution is to look for mobile apps that have exactly the same visual appearance. To avoid such detection, the adversary can simply create a slightly modified version of the spoofing screen. For example, small changes in the login screen element positions are hard to notice and are unlikely to retain the user from entering her login credentials. Consequently, this approach would fail to catch many spoofing attacks. Such visually modified attacks are observed in the wild. For example, the Android.Fakeneflic malware [176], discovered on Google's Android market, impersonated the legitimate Netflix application with minor visual modifications (Figure 5.1). Such attacks would not be detected by a simple comparison scheme that looks for an exact visual match. To summarize, we do not focus on detection of perfect copies, as such detection is easy to avoid, and spoofing apps seen in the wild often show minor visual differences. The adversary has an incentive to introduce enough visual change to evade simple detection, but not enough for users to become alarmed. The primary contribution of this chapter is to explore this space; to determine how much change do users tolerate.

The second straw-man solution is to flag all applications that have high similarity to a reference application, with regards to a common image similarity metric, e.g., converting a screenshot to gray-scale, and scaling it down to a fixed size (64×64 pixels). Comparing such thumbnails by pixel difference is tolerant to many minor visual modifications. For example, screenshots with change of colors, or other minor pixel differences, would be deemed highly similar, and the metric would detect such spoofing attacks. However, the metric would fail on more complex examples (Figure 5.2), as it does not capture the visual properties that users consider relevant. As our user study shows (Section 5.4), many screens are perceived as similar

Figure 5.2: *Examples of simple (changing background color) and more complex spoofing (repositioning elements).*

by users, even though the screens are *very dissimilar* in terms of their pixel values. For example, many users mistook a pink Facebook screen with perturbed element positions as genuine. Such advanced spoofing would not be caught by the above simple metric — for robust detection more sophisticated techniques are needed.

In this chapter we explore visual similarity as perceived by the users. We take a different approach and design a spoofing detection system that estimates how many users would fall for a spoofing attack. We use *deception rate* as a novel similarity metric that represents the estimated attack success rate. Given two screenshots, one of the examined app and one of the protected reference app, our system (Figure 5.3) estimates the percentage of users that would mistakenly identify the examined app as the reference app (deception rate). This estimation is done by leveraging results from a study on how users perceive visual similarity on mobile app user interfaces. The deception rate can be seen as a risk measure that allows our system to determine if the examined application should be flagged as a potential spoofing application. An example policy is to flag any application where the deception rate exceeds a threshold.

Deception rate is a conceptually different similarity metric from the ones previously proposed for similarity analysis of phishing websites. These works extract structural [12,111,154,205,208] as well as visual [43,72,125] similarity features and combine them into a similarity score that alone is not expressive, but enables comparison to known attack samples [111, 129]. The extracted features can also be fed into a system that is trained using known malicious sites [72, 194, 205]. Such similarity metrics are interpreted with respect to known attacks, and may not be effective in detecting previously unseen spoofing attacks.

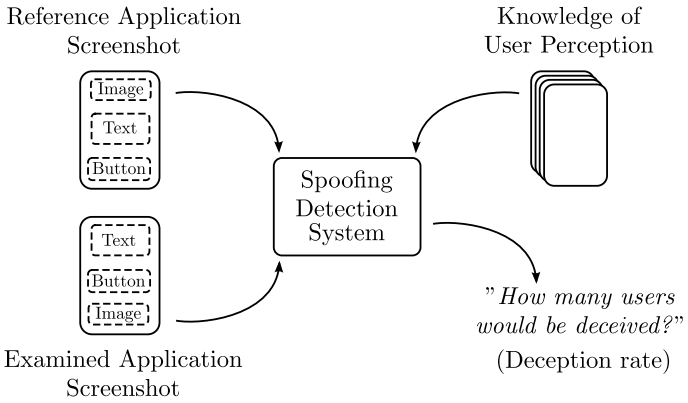


Figure 5.3: *Approach overview. The spoofing detection system takes as inputs screenshots of a reference app and an examined app. Based on these screenshots and knowledge on mobile application user perception, the system estimates deception rate for the examined app.*

Deception rate has different semantics, as it captures the *perceived similarity* of spoofing screens. For example, a mobile app login screen where elements have been reordered may have different visual features but, as our user study shows, is perceived similarly by many users. Deception rate estimates how many people would mistakenly identify the spoofing app as the genuine one (risk measure) and, contrary to previous similarity metrics, is applicable also in scenarios where a large dataset of known spoofing samples are not available. We emphasize that our system is complementary to existing approaches, and that realization of such a system requires good understanding of what type of mobile app interfaces users perceive as similar and what type of visual modifications users are likely to notice. This motivates our user study, the results of which we describe in Section 5.4.

5.3.2 Case Study: Login Screen Spoofing

We focus on spoofing attacks against mobile application login screens, as they are the most security-sensitive ones in many applications. We examined the login screens of 230 different apps and found that they all follow a similar structure. The login screen is a composition of three main elements: (1) the logo, (2) the username and password input fields, and (3) the login button. Furthermore, the login screen can have additional, visually less salient elements, such as a link to request a forgotten password

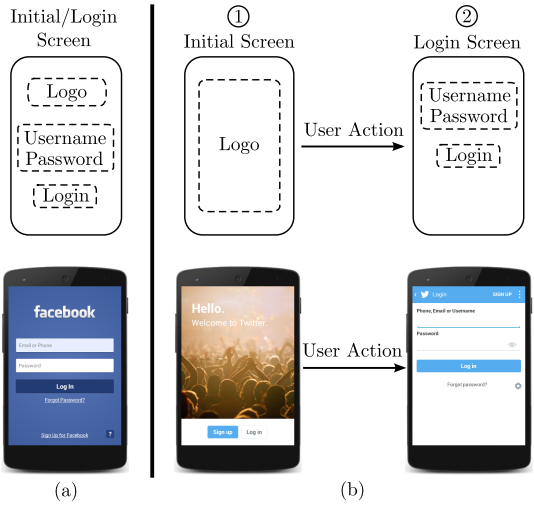


Figure 5.4: *Model for mobile app login screens. The login screen has three main elements: logo, username and password input fields, and login button. The login functionality is either (a) standalone or (b) distributed.*

or register a new account. Some mobile apps distribute these elements across two screens: the first (initial) screen contains the logo, or a similar visual identifier, as well as a button that leads to the login screen, where the rest of the main elements reside.

The common structure of mobile app login screens enables us to model them, and their simple designs provide a good opportunity to experiment on user perception. Mobile app login screens have fewer modification dimensions to explore, as compared to more complex user interfaces, such as websites. Throughout this work we use the login screen model illustrated in Figure 5.4 that captures both standalone and distributed logins screens. Out of the 230 apps we examined, 136 had a standalone login screen, while 94 had a distributed one. All apps conformed to our model. We experiment on user perception with respect to this model, as the adversary has an incentive to create spoofing screens that resemble the legitimate login screen. Our study confirms this assumption.

5.3.3 Attacker Model

We assume a strong attacker capable of creating arbitrary spoofed login screens, including login screens that deviate from our model. We distinguish

between two spoofing attack scenarios regarding user expectations and goals. In all the spoofing attacks listed in Section 5.2, the user's intent is to access the targeted application. This implies that the user expects to see a familiar user interface and has an incentive to log in. The adversary could also present a spoofing screen unexpectedly, when the user is not accessing the target application. In such cases, the user has no intent, nor similar incentive, to log in. We focus on the first case, as we consider such attacks more likely to succeed.

We assume an attacker that controls a malicious spoofing app running on the user smartphone. Besides the spoofing screen, the attacker-controlled app appears to the user as entirely benign (e.g., a game). The attacker can construct the spoofing screen statically (e.g., using Android manifest files) or dynamically (e.g., creating widgets at runtime). In both cases, the operating system is aware of the created element tree, a structure similar to DOM trees in websites. The attacker can draw the screen pixel by pixel, in which case the operating system sees only one element, a displayed picture. The attacker can also exploit the properties of human image perception. For example, the attacker can display half of the spoofed screen in one frame, and the other half in the subsequent frame. The human eye would average the input signal and perceive the complete spoofing screen.

5.4 Change Perception User Study

Visual perception has been studied extensively in general, and prior studies have shown that users are surprisingly poor at noticing changes in successively shown images (change blindness) [151,168]. While such studies give us an intuition on how users might notice, or fail to notice, different login screen modifications, the results are too generic to be directly applied to the spoofing detection system outlined above. User perception of visual change in mobile app user interfaces has not been studied thoroughly before.

We conducted a large-scale online study on the similarity perception of mobile app login screens. The purpose of this study was three-fold: we wanted to (1) understand the effect of different types of visual login screen modifications, (2) gather training data for the spoofing detection system, and (3) gain insights that could aid us in the design of our system. The study was performed as online surveys on the crowd-sourcing platform CrowdFlower. The platform allows creation of online jobs that human participants perform in return of a small payment. In each survey, the participants evaluated a single screenshot of a mobile app login screen by answering questions (see Appendix A).

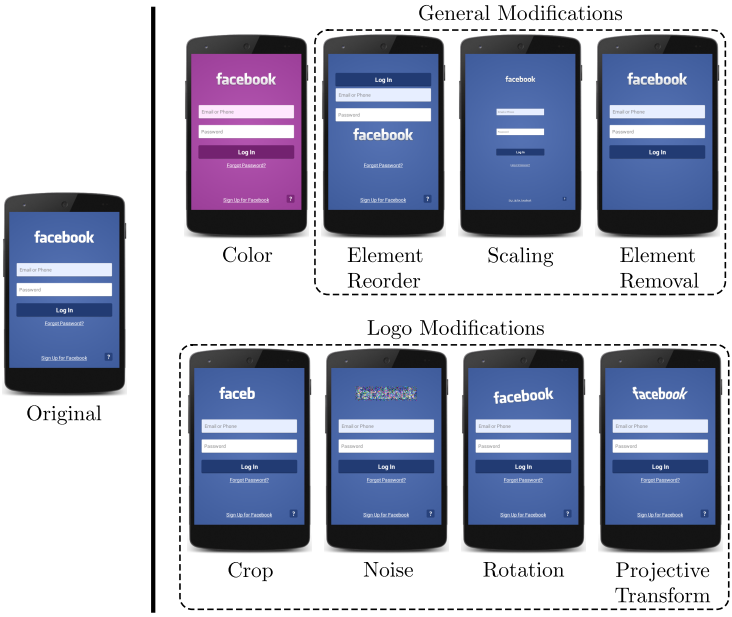


Figure 5.5: *Examples of Facebook login screen spoofing samples. The original login screen is shown on the left. We show an example of each type of visual modification we performed: color, general modifications, and logo modifications.*

We first performed an initial study, where we experimented with visual modifications on the Android Facebook application. We chose Facebook, as it is a widely used application. After that, we carried out follow-up studies where we tested similar visual modifications on Skype and Twitter apps, as well as combinations of visual changes. Below, we describe the Facebook study and summarize the results of the follow-up studies. We did not collect any private information about our study participants. The ethical board of our institution reviewed and approved our study.

5.4.1 Sample Generation

A *sample* is a screenshot image presented to a study participant for evaluation. We created eight datasets of Facebook login screens, and in each dataset we modified a single visual property. The purpose of these datasets was to evaluate how users perceive different types of visual changes as well

as to provide training data for the spoofing detection system. Figure 5.5 illustrates each performed modification:

- *Color modification.* We modified the hue of the application login screen. The hue change affects the color of all elements on the login screen and the dataset contained samples representing uniform hue changes over the entire hue range.
- *General modifications.* We performed three general modifications on the login screen elements. (1) We reordered and (2) scaled down the size of the elements. We did not increase the size of the elements, as the username and the password fields are typically full width of the screen. Furthermore, (3) we removed any extra elements from the login screen.
- *Logo modifications.* We performed four modifications on the logo: we (1) cropped the logo to different sizes, taking the rightmost part of the logo out, (2) added noise of different intensity, (3) rotated the logo both clockwise and counterclockwise, and (4) performed projective transformations on the logo.

We created synthetic spoofing samples as no extensive mobile spoofing app dataset is available. While the chosen modifications cover some known spoofing attacks (e.g., Figure 5.1), they are certainly not exhaustive, as the attacker can change the interface in many different ways, e.g., adding different background images, replace logo with text. The goal of our work is not to optimize the system for the detection of known attacks, but rather to create a system that is able to detect also previously unseen spoofing screens. The sample set could be extended in many ways, but a single user study cannot cover all possible modifications.

5.4.2 Recruitment and Tasks

Participant recruitment. We recruited test participants by publishing survey jobs on the crowd sourcing platform. An example survey had a title *"Android Application Familiarity"* and the description of the survey was *"How familiar are you with the Facebook Android application?"*. We specified in the survey description that the participant should be an active user of the tested application, and we recruited 100 study participants for each sample, accepted participants globally, and required the participants to be at least 18 years old. The study participants were allowed to evaluate multiple samples from different datasets, but only one sample from each dataset.

Unique study participants	2,910
Participants that completed multiple surveys	1,691
Screenshot samples	59
Total evaluations	5,900
Accepted evaluations after filtering	5,376
Average number of accepted evaluations per sample	91

Table 5.1: *Statistics of the Facebook user study.*

Age	
18-29	55.12%
30-39	29%
40-49	11.82%
50-59	3.33%
60 or above	0.72%

Gender	
Male	72.54%
Female	27.45%
Education	
Primary school	2.06%
High school	34.57%
Bachelor	63.36%

Table 5.2: *Demographics of the Facebook user study.*

For example, a study participant could complete two surveys: one where we evaluated color modification samples and another regarding logo crop, but the same participant could not complete multiple surveys on color modification. In total 2,910 unique participants evaluated 5,900 Facebook samples. Our study statistics and participant demographics are listed in Table 5.1 and Table 5.2.

Study tasks. Each survey included 12 to 16 questions. We asked preliminary questions on participant demographics, tested application usage frequency, and a control question with a known correct answer. We showed the study participant a sample login screen screenshot and asked the participant the following questions: *"Is this screen (smart phone screenshot) the Facebook login screen as you remember it?"* and *"If you would see this screen, would you login with your real Facebook password?"*. We provided *Yes* and *No* reply alternatives on both questions. Using the percentage of *Yes* answers, we compute *as-remembered rate* and *login rate* for each sample. We also asked the participants to comment on their reason to log in or retain from logging in. For the interested reader, we provide the full list of questions in Appendix A.

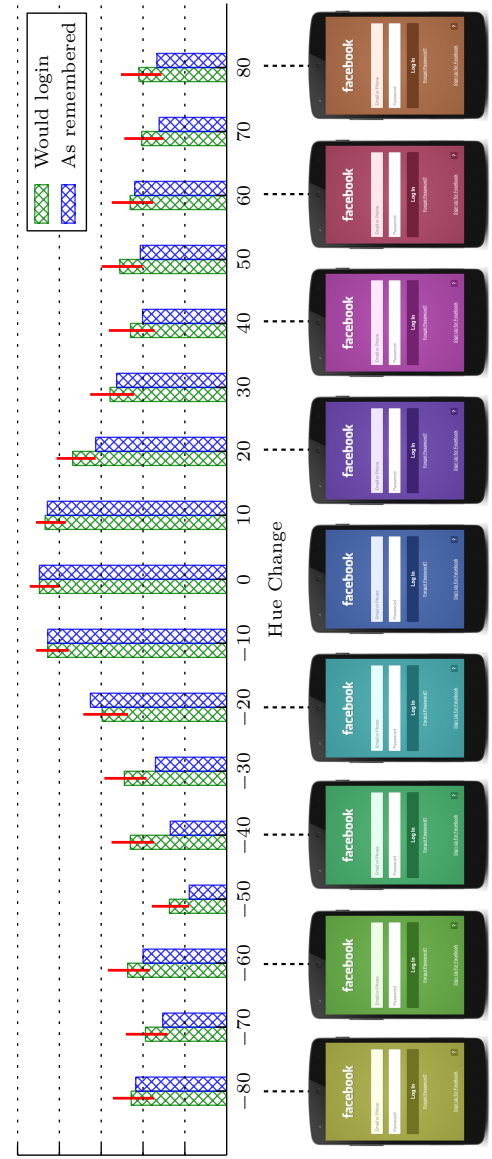


Figure 5.6: *Color modification results. We illustrate the percentages of users that perceived a Facebook login screen sample with modified color as genuine (as-remembered rate) and would login to the application if such a screen is shown (login rate). Color has a significant effect on both rates.*

Through the chosen design of our user study, we purposefully primed the participants to expect to see a login screen of the studied apps. We simulate the setting in which the user wants to login, but is presented with a login screen that is different than the user remembers.

5.4.3 Results

We discarded survey responses where the participants did not indicate active usage of the Facebook app or gave an incorrect reply to the control question. After filtering, we had 5,376 completed surveys and, on the average, 91 user evaluations per screenshot sample.

Color modification. The color modification results are illustrated in Figure 5.6. We plot the observed login rate in green and the as-remembered rate in blue for each evaluated sample. The red bars indicate bootstrapped 95% confidence intervals. We performed a chi-square test of independence with a significance level of $p = 0.05$ to examine the relation between the login responses and the sample color. The relation between these variables was significant ($\chi^2(16, N = 1551) = 194.44), p < 0.001$) and the study participants were less likely to log in to screens with high hue change. When the hue change is maximal, approximately 40% of the participants indicated that they would still log in. For several samples we noticed slightly higher login rate compared to as-remembered rate. This may imply that some users were willing to log in to an application, although it looked different from their recollection. We investigated reasons for this behavior from the survey questions and several participants replied that they noticed the color change, but considered the application genuine nonetheless. One participant commented: *"Probably Facebook decided to change their color."* However, our study was not designed to prove or reject such hypothesis.

General modifications. The general element modification results are shown in Figure 5.7. Both element reordering ($\chi^2(5, N = 546) = 15.84, p = 0.007$) and scaling ($\chi^2(9, N = 916) = 245.56, p < 0.001$) had an effect on the observed login rates. Samples with scaling 50% or less showed login rates close to the original, but participants were less likely to login to screens with high scaling. This could be due to users' habituation of seeing scaled user interfaces across different mobile device form factors (e.g., smartphone user interfaces scaled for tablets). One participant commented his reason to login: *"looks the same, just a little small."* When the elements were scaled more than 50%, the login rates decreased fast. At this point the elements became unreadably small. Removal of extra elements

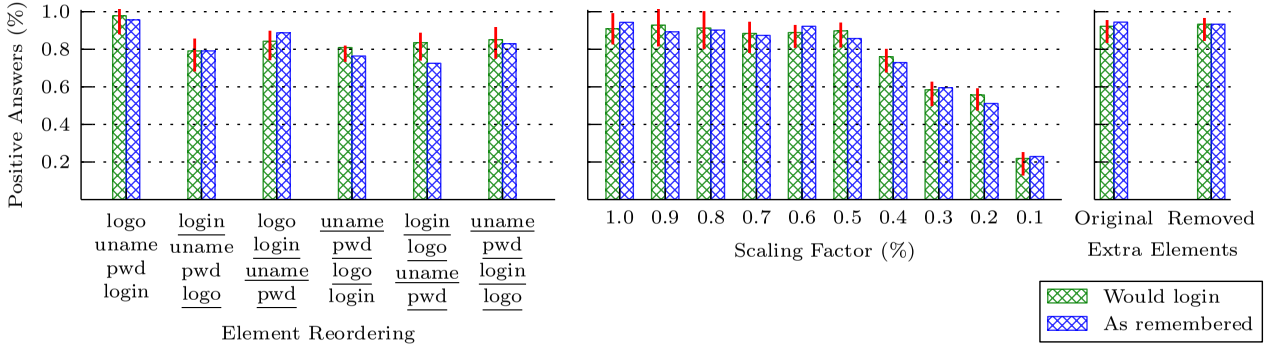


Figure 5.7: *General modifications results. Percentages of users that perceived a Facebook login screen sample with general modifications as genuine and would login. Element reordering modification had a small but statistically significant effect, scaling caused a significant effect, and extra element removal showed no effect.*

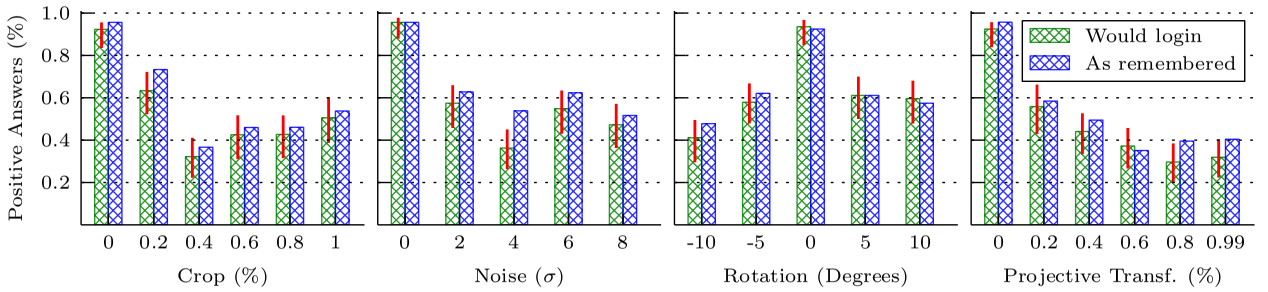


Figure 5.8: *Logo modifications results. Percentages of users that perceived a Facebook login screen sample with logo modifications as genuine and would login to the application. All logo modifications caused a significant effect.*

(forgotten password or new account link) had no effect on the login rate ($\chi^2(1, N = 180) = 0.0, p = 1.0$).

Logo modifications. The logo modification results are shown in Figure 5.8. The relation between the login rate and the amount of crop was significant ($\chi^2(5, N = 540) = 83.75, p < 0.001$). Interestingly, we noticed that the lowest login rate was observed at 40% crop. This implies that the users may find the login screen more trustworthy when the logo is fully missing compared to seeing a partial logo.

The amount of noise in the logo had an effect on login rates ($\chi^2(4, N = 460) = 75.30, p < 0.001$), as users were less likely to log in to screens with noise. Approximately half of the study participants answered that they would login even if the logo was unreadable due to noise. This result may imply habituation to software errors and one of the participants commented the noisy logo: *"I would think it is a problem from my phone resolution, not Facebook."* Participants were less likely to log in to screens with a rotated logo ($\chi^2(4, N = 462) = 57.25, p < 0.001$) or a projected logo ($\chi^2(5, N = 542) = 102.45, p < 0.001$).

Conclusions. The experimented eight visual modifications were perceived differently. While some modifications caused a predominantly systematic pattern (e.g., color), in others we did not notice a clear relation between the amount of the modification and the observed login rate (e.g., crop). One modification (extra element removal) caused no effect. We conclude that the system should be trained with samples that capture various types of visual modifications.

5.4.4 Follow-up Studies and Study Method

We performed similar studies for the Skype and Twitter apps. Skype results were comparable to those of Facebook. Twitter app has a distributed login screen and we noticed different patterns than in the previous two studies. Additionally, we evaluated combinations of two and three visual modifications. In total we collected 34,240 user evaluations from 5,438 unique study participants, and we used the cumulative collected data to train our spoofing detection system.

We measured login rates by asking study participants questions in contrast to observing participants under login operation. We chose this approach to allow large-scale data collection for thousands of sample evaluations. Participants in our study were allowed to evaluate multiple samples from different datasets which may have influenced the results.

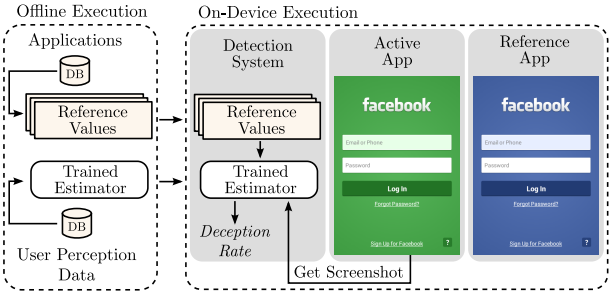


Figure 5.9: *Detection system overview. The system pre-processes legitimate apps offline (e.g., at the marketplace) to obtain reference values, and trains an estimator. On the user's device, the system periodically extracts screenshots and estimates their deception rate.*

5.5 Spoofing Detection System

Through our user study we gained insight into what kind of visual modifications users notice, and more importantly, fail to notice. In this section we design a spoofing detection system that leverages this knowledge. We instantiate the system for Android, while many parts of the system are applicable to other mobile platforms as well.

5.5.1 System Overview

Our system is designed to protect *reference applications*, i.e., legitimate apps with login functionality. The goal of our system is to, given a screenshot, estimate how many users would mistake it for one of the known reference apps. The system (Figure 5.9) consists of two parts: a training and preprocessing component that runs on the market and a runtime detection system on users' phones. On the market, each reference app's login screen is detected, pre-processed, and a deception rate estimator is trained using the user perception data from our user study. The analyzed login screens serve as the *reference values* for the on-device detection.

On the device, the system periodically extracts a screenshot of the currently active app. We analyze screenshot extraction rates needed for effective detection in Section 5.8. Each extracted screenshot is analyzed using the estimator with respect to the reference values of the protected apps. Both the trained estimator and the reference values are downloaded from the market (e.g., upon installing an app). The system outputs a

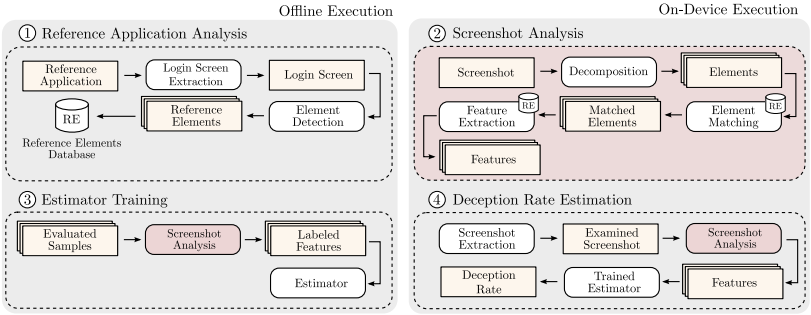


Figure 5.10: *Detection system details. The system consist of four main components: reference app analysis, screenshot analysis, estimator training and deception rate estimation.*

deception rate for each analyzed screenshot, with respect to each protected app. The deception rates can be used to warn the market or the user.

The apps that should be protected (i.e., labeled as reference apps), can be determined in multiple ways: the user can choose the apps that require protection, the system can automatically select the most common spoofing targets (e.g., Facebook, Skype, Twitter), or all installed apps with login functionality. We focus on the approach where the protected apps are chosen by the user. A complete view of the system is illustrated in Figure 5.10, and we proceed by describing each system component.

5.5.2 Reference Application Analysis

Our system protects reference apps from spoofing. To analyze an extracted screenshot with respect to a reference value, we first obtain the reference app login screen and identify its main elements (reference elements) according to our login screen model (Figure 5.4). We assume reference app developers that have no incentive to obfuscate their login screen implementations. On the contrary, developers can be encouraged to mark the part of the user interface (activity) that contains the login screen that should be protected. The reference app analysis is a one-time operation performed, e.g., at the marketplace on every app update, and its results distributed to the mobile devices. To find the activity that represents the login screen, we developed a tool that automatically explores a specified application and stores any found login screens. From the login screens, the tool detects and stores *reference elements* into a tree structure.

5.5.3 Screenshot Analysis

The goal of the screenshot analysis is to, given the screenshot of the examined application, as well as the reference elements, produce suitable features for deception rate estimation and estimator training. The screenshot analysis includes three operations: decomposition, element matching, and feature extraction, as shown in Figure 5.10.

Decomposition. Mobile application UIs commonly exhibit a clean and simple design, when compared to more complex ones, e.g., web sites. Such design simplicity enables us to efficiently split the screenshot into constituent elements, and we illustrate the steps in Figure 5.11. To identify element borders we perform a set of image processing steps, including edge-detection, dilation, closure and gradient.

Element matching. The next step is to match the detected elements to the reference elements. To find the element that is the closest match to the reference logo, we use the ORB feature extractor [155]. While SIFT extractors [115] have been successful in detecting logos in natural images [157], we found SIFT to be ill-suited for mobile app logos, especially in cases where only partial (cropped) logos were present. We compute ORB keypoints over the reference logo as well as the whole examined screenshot and we match the two sets. The element that matches with the most keypoints, and exceeds a minimum point density threshold, is declared as the logo. For the remaining elements, we perform template matching to every reference element (username field, password field, login button), on different scaling levels. Keypoint extraction is generally not effective, as the login screen elements are typically simple, and have few keypoints. After these steps, we have a mapping between the examined and reference elements.

Feature extraction. Once the elements are matched, we extract two common visual features (color and element scaling) and more detailed logo features, as users showed sensitivity to logo changes. The extracted features are relative, rather than absolute, as their values are computed with respect to the reference elements or entire reference screen. We explain our features below:

1. *Hue.* The difference between the average hue value of the examined screenshot and the reference screen.
2. *Element Scaling.* The ratio of minimum-area bounding boxes between all reference and examined elements, except the logo.

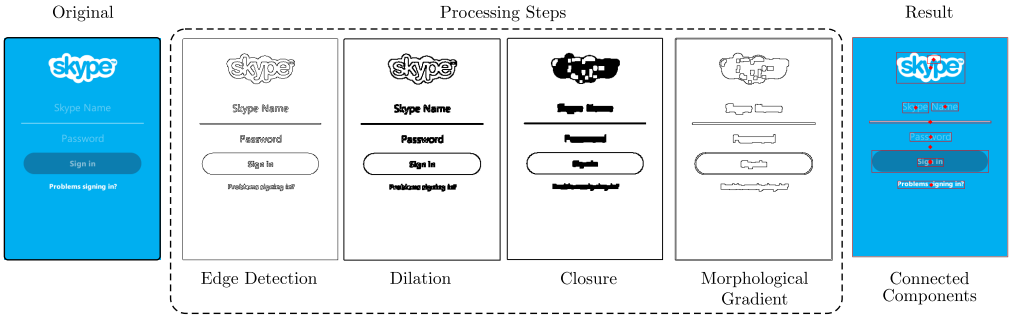


Figure 5.11: *Decomposition process. The processing steps in the middle includes common image analysis techniques. The final step is a connected components algorithm and filtering of smaller regions. For visual clarity, we inverted the colors in the processing steps.*

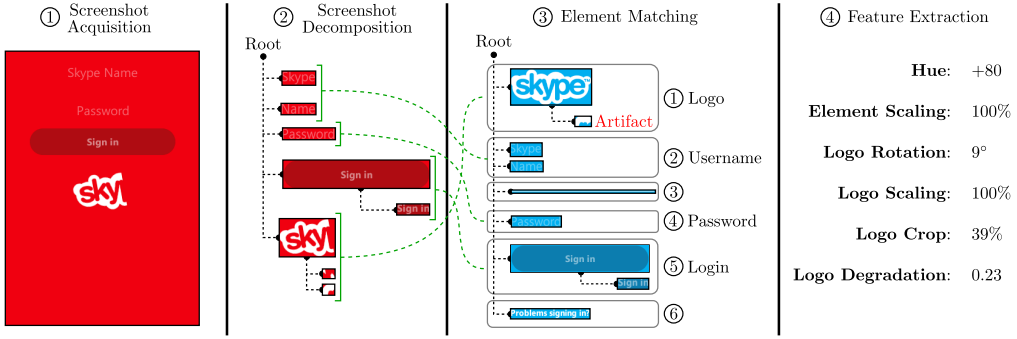


Figure 5.12: *Summary of the screenshot analysis. (1) The starting point is a mobile application login screenshot. (2) We decompose the screenshot to a tree hierarchy. (3) We match the detected elements to reference elements. (4) We extract features from the detected elements with respect to the reference elements.*

3. *Logo Rotation.* The difference between the angles of minimum-area bounding boxes of the examined and reference logos.
4. *Logo Scaling.* We perform template matching between the examined and reference logos at different scales and express the feature as the scale that produces the best match.
5. *Logo Crop.* We calculate the amount of logo crop as the ratio of logo bounding box areas.
6. *Logo Degradation.* As precise extraction of logo noise and projection is difficult, we approximate similar visual changes with a more generic feature that we call logo degradation. Template matching algorithms return the position and the minimum value of the employed similarity metric and we use the minimum value as the feature.

We illustrate the element matching and feature extraction steps in Figure 5.12. In cases where no logo was identified in the matching phase, all logo features are set to null (except logo crop which is set to 100%). Our analysis is designed to extract features from screenshots that follow our login screen model. Many of these features (color change, scaling) are seen known in spoofing apps (Android.Fakeneflic).

5.5.4 Training and Deception Rate Estimation

The detection system is trained once, using the available user perception data from our user study, and subsequently used for all apps. We extract features from every sample of the study and augment the resulting feature vectors with the observed login rate. In feature extraction, as the reference value we use the unmodified login screen of the app that the sample represents. As deception rate (i.e., the percentage of users that would confuse the examined screenshot with the reference app) is a continuous variable, we estimate it using a regression model. Training can be performed offline for each reference app separately.

Deception rate estimation is performed on the device at detection system runtime. As shown in Figure 5.10, the extracted screenshot is first analyzed. The decomposition phase of the analysis is performed once, and the rest of the analysis steps are repeated for each reference app. The extracted features are used to run the trained estimator. The result of the estimation operation is a set of deception rates, one for each protected app. If any of the deception rates exceeds a threshold value, one or more possible spoofing apps have been found and their identities can be communicated to the application marketplace or the user can be warned.

5.5.5 Implementation

We implemented the reference application analysis tool as a modified Android emulator environment. Similar analysis can be implemented by instrumenting the reference application, but we modified the runtime environment to support the analysis of native applications as well. We implemented the remaining offline tools as various Python scripts using the OpenCV [34] library for image processing and scikit-learn for estimator training. The on-device detection system can be implemented in multiple ways, including a modification to the Android runtime or as a standalone application. For ease of deployment, we implemented the on-device components as a regular Android (Java) app using OpenCV.

5.6 Evaluation

In this section, we evaluate the estimation accuracy and the runtime performance of the detection system.

5.6.1 Accuracy Evaluation

Reference Application Analysis Accuracy. We evaluated the accuracy of our reference app analysis tool (Section 5.5.2) on 1,270 apps, downloaded from Google Play and other marketplaces. The tool reported 572 potential login screens. Through manual verification, we found 230 login, 153 user registration, and 77 password change screens. The remaining 120 screens contained no login functionality, and those we classify as false positives.

We manually verified 50 random apps from the set of 698 apps our tool reported as not having a potential login screen. We found 3 false negatives due to an implementation bug that was since fixed. We conclude that the tool can effectively find all login screens that require protection. The tool provides an over approximation, but a small number or false positives does not hamper security, as they only introduce additional reference values for similarity comparison. Moreover, developers have an incentive to help the reference login screen detection and they can explicitly mark which activity constitutes the login screen for even more accurate detection.

Decomposition Accuracy. To evaluate the accuracy of our screenshot decomposition algorithm, we decomposed 230 login screen screenshots. We manually verified the results and found that we detected all login screen elements correctly on 175 screens. We found 29 screens that correctly decomposed all but one element, and 9 screens with correct decompositions for all but two elements. Our algorithm failed to decompose 18 screens.

Figure 5.13: *Decomposition examples. The login screen decomposition algorithm works well in practice. We outline in red the borders of detected elements, while the red diamond represent element centroids. Some login screens (tumblr, last screenshot) are visually complex and are inherently hard for our approach to analyze.*

Certain types of login screens are challenging for our approach. For example, the login screen of the Tumblr application contained a blurred natural image in the background, and our algorithm detected many erroneous elements (see Figure 5.13). Our current implementation is optimized for clean login screens, as those are the pre-dominant login screen types. The majority (92%) of analyzed screenshots were visually simple and decomposed. We discuss noisy spoofing screens as a possible detection avoidance technique in Section 5.8.

5.6.2 Estimation Accuracy

To evaluate the deception rate estimation accuracy, and to demonstrate the feasibility of this approach, we trained our detection system using the results of our user study (a deployed system would, of course, be trained with more data). Our total training data consists of 316 user-evaluated samples of visual modifications and each sample was evaluated either by 100 (single modification) or 50 (two and three modifications) unique users.

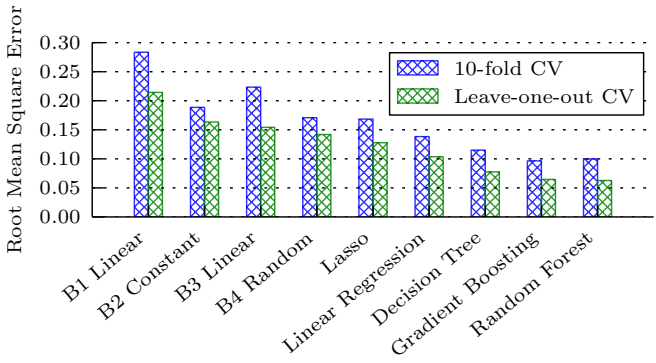


Figure 5.14: *Deception rate accuracy. Evaluation of five regression and four baseline models (B1–B4) trained on the combined datasets of Facebook and Skype. The random forest regressor performs the best.*

We omitted training samples that express visual modifications that our current implementation is unable to extract (e.g., noise).

We experimented with several regression models of different complexities and trained two linear models (Lasso and linear regression), a decision tree, as well as two ensemble learning methods (gradient boosting, random forests). To compare our detection accuracy to straightforward approaches, we use four baseline models out of which the latter two utilize prior knowledge obtained from our user study:

- *B1 Linear.* The deception rate drops linearly with the amount of visual modification from 1 to 0.
- *B2 Constant.* The deception rate is always 0.75.
- *B3 Linear.* The deception rate drops linearly with the amount of visual modification from 1 to 0.2. Login rates stayed predominantly above 20% in our study.
- *B4 Random.* The deception rate is a random number in the the most observed range in our study (0.3–0.5).

To estimate the deception rate, we extract features from the analyzed screenshot with respect to a reference app and we feed the feature vector to the trained regressor. The estimator outputs a deception rate that can be straightforwardly converted into a spoofing detection decision. We performed two types of model validation: leave-one-out and 10-fold cross-validation. We report the results in Figure 5.14 and we observe that the

more complex models perform significantly better than our baseline models. The best model was random forest, with a root mean square (RMS) error of 6% and 9% for the leave-one-out and 10-fold cross validations respectively (95% of the estimated deception rates are expected to be within two RMS errors from their true values). The low RMS values show that a system trained on user perception data can accurately estimate deception rates for mobile application spoofing attacks.

The detection system should estimate deception rate accurately even for applications it did not encounter before. To evaluate the estimation accuracy of attacks that target apps that were not present in the training data, we trained a random forest regressor using Facebook samples, and evaluated it on Skype samples, and vice-versa. We observed an RMS error of 13% in both cases. When samples from the spoofing target application are not part of the training dataset, the estimation accuracy decreases slightly. We conclude that our system is able to accurately estimate deception rate in the tested scenarios, even if the target app is not part of the training data. Our training set has limited size and with more extensive training data we expect even better accuracy.

To evaluate false negatives of our system, we estimated the deception rates of various screenshots that we extracted by crawling the user interfaces of randomly chosen mobile apps, with regards to the Facebook reference login screen. Due to the large difference between the login screens, as expected, all screenshots reported very low deception rates. We do not provide a ROC analysis, as it would require a significant dataset of spoofing apps. At the moment such a dataset does not exist.

5.6.3 Performance Evaluation

We evaluated the performance of the on-device screenshot analysis and deception rate estimation. For the offline (marketplace) components we only evaluated accuracy, as those are fast and are not time-critical operations. We measured the performance on three devices: an older smartphone model (Samsung Galaxy S2) and two more recent devices (Nexus 5 and Nexus 6). Averaged over 100 runs, a single reference app comparison takes 183 ± 28 ms (Nexus 5), 261 ± 26 ms (Nexus 6) and 407 ± 69 ms (Galaxy S2). The process scales linearly with the number of protected apps: the decomposition of the extracted screenshots is performed once, and the remaining analysis steps are repeated for each reference value. Assuming five protected apps, the complete analysis takes 667 ms (Nexus 5). We argue that the number of apps requiring protection would be low, as the majority of apps running on the phone are commonly not security-sensitive.

	Galaxy S2	Nexus 5	Nexus 6
Screenshot extraction	10 ± 3 ms	21 ± 13 ms	19 ± 7 ms
Decomposition	99 ± 19 ms	41 ± 10 ms	42 ± 8 ms
Element matching	147 ± 35 ms	54 ± 16 ms	106 ± 16 ms
Feature extraction	150 ± 34 ms	67 ± 12 ms	94 ± 13 ms
Estimator	0.5 ± 0.9 ms	0.1 ± 0.3 ms	0.4 ± 0.5 ms
Total	407 ± 69 ms	183 ± 28 ms	261 ± 26 ms

Table 5.3: *Performance evaluation of our implementation.*

The detection system extracts and analyzes screenshots only when an untrusted (i.e., not whitelisted) app is active. For example, the platform provider can whitelist popular apps from trusted developers (Facebook, Twitter, Whatsapp). The detection system can also perform a less expensive *pre-filtering* operation to determine, and only proceed with the full analysis, if the examined screenshot vaguely resembles a login screen. We leave development of such pre-filtering mechanisms as future work.

The on-device performance primarily depends on the size of the analyzed screenshot. Modern smartphones have high screen resolutions (e.g., 1080×1920) and analyzing such large images is expensive and does not increase system accuracy. It is important to note that screenshot extraction time depends only on the output screenshot resolution and not on the physical screen resolution itself. For all our measurements we extracted screenshots of size 320×455 pixels as the resolution provides a good ratio of element detection accuracy and runtime performance. Our initial experiments show that the image resolution (and with it, execution time) can be decreased even further, and determining the optimal resolution we leave as future work.

5.7 Detection Probability Analysis

In this section, we explain how often screenshots can be extracted on the device, given a pre-defined amount of allocated system resources. If a spoofing attack takes place, we analyze the probability that at least one screenshot of the spoofing application is captured. We also present a collaborative detection model that enables significantly fewer screenshot analysis operations per device.

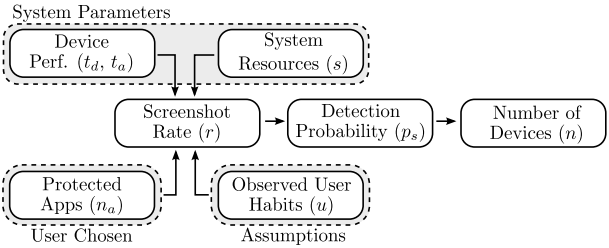


Figure 5.15: *Analysis intuition. System parameters, user behavior assumptions, and a user-chosen number of protected apps define the screenshot rate, the detection probability for a single spoofing attack, and the number of devices required for effective collaborative detection.*

5.7.1 Detection Probability on a Single Device

In Figure 5.15, we illustrate the intuition of our analysis. The system has two controllable parameters: the share of the system resources s that are allocated for spoofing detection and the number of reference apps n_a the system protects. Together with device performance and the observed user habits (the share of time spent on unknown apps u), these two parameters define the screenshot rate r which in turn determines the detection probability for a single spoofing attempt p_s, as well as the number of devices n needed for efficient collaborative detection. In what follows, we introduce the rest of the terms gradually and, for ease of reference, summarize our terminology in Table 5.4.

In a typical deployment, the share of system resources allocated for the detection system would be chosen by the platform provider. For our analysis, we use $s = 1\%$, as we assume that one percent overhead does not hinder user experience nor overly drain the battery. The number of protected applications is chosen by the user. We assume that in most cases the user would choose to protect a small number of important services (e.g., banking, e-mail, Facebook, Skype, Twitter) and use the value $n_a = 5$ for our analysis.

For analysis simplicity, we assume that the user spends a constant time t_l on the spoofed login screen. In a recent study [121], users spent 4 – 28 seconds on the login screen, so $t_l = 5$ seconds is a safe assumption. We also assume that the user spends a constant share u of her time on unknown (non-whitelisted) apps. According to [49], smartphone users spend 88% of their time on five of their favorite apps, so setting $u = 0.25$

System Presets	s	Share of allocated system resources
	t_d	Decomposition time (device perf.)
	t_a	Analysis time (device perf.)
Observed User Habits	t_l	Time spent on login screen
	u	Share of time spent on unknown apps
User Chosen	n_a	Number of protected applications
Detection Properties	r	Screenshot rate
	p_s	Detection probability, single spoofing
	p	Detection prob., collaborative system
	n	Number of devices with spoofing app

Table 5.4: *Summary of analysis terminology.*

is a safe assumption. The detection system can monitor the runtime usage of unknown apps and adjust a user-specific u accordingly.

For device performance, we use the values from our implementation evaluation on Nexus 5, where the analysis time of a single screenshot is approximately 180 ms. The screenshot extraction and decomposition time t_d is approximately 60 ms, while the remaining screenshot analysis time t_a that needs to be repeated for each reference app is approximately 120 ms. Using such device performance, system parameters and analysis assumptions, we compute the screenshot rate r as follows:

$$r = \frac{u}{s}(t_d + n_a t_a) \approx 16.5 \text{ s}$$

That is, given 1% of allocated system resources, a screenshot can be analyzed on average once per 16.5 seconds when an unknown app is active.

The detection probability for a single spoofed login screen p_s is the probability that, when a spoofed login screen is shown to the user for $t_l = 5$ seconds, the detection system captures, and analyzes, at least one screenshot during that time. To avoid simple detection evasion where the malware never shows spoofed screens at pre-determined screenshot extraction times, we assume that screenshots are taken at random points in time, according to the chosen screenshot rate. Given the randomized screenshot extraction model, we model p_s as a random number from the Poisson distribution $P(x;\mu)$, where x is the number of successes in a given time period (zero successes means that no screenshots are taken in the time period) an μ is the mean of successes in the same time period. The number of screenshots taken on the average can be calculated as t_l/r (e.g.,

5/16.5 in our example scenario). The detection probability p_s becomes:

$$p_s = 1 - P(0, \frac{t_l}{r}) \approx 0.26$$

We observe that the probability of detecting a single spoofed login operation is low. Moreover, the adversary does not have an incentive to repeat a successful attack on the same device. Once the user's login credentials have been stolen, the malicious app can, e.g., remove itself. For these reasons we focus on a more effective collaborative deployment model that leverages the *many eyes principle*.

5.7.2 Collaborative Detection

An instance of the detection system can be running on a large number of devices (e.g., all devices from the same platform provider), where each device takes screenshots at random points in time, according to the chosen screenshot rate. When one of the devices finds a potential spoofing login screen, the identity of the application is reported to the platform provider (or the app marketplace) which can examine the application and remove it from all of the devices, if confirmed malicious. For analysis simplicity, we assume that all participating devices have similar performance and use the same, previously chosen system parameters, but deployments where devices are configured differently are, of course, possible. The detection probability p of the collaborative system, i.e., the probability that at least one device will detect the spoofing attack, is defined as:

$$p = 1 - (1 - p_s)^n$$

where n is the number of devices infected with the spoofing app. Assuming our example parameters, to reach detection probability $p = 0.99$, we need the malicious application to be installed and active on only 16 devices:

$$n = \lceil log_{1-p_s}(1-p) \rceil = 16$$

Spoofing apps that infected thousands of devices have been reported [69], so we consider this a very low number for common wide-spread attacks that target globally used apps, such as Facebook, Skype or Google. Figure 5.16 illustrates the detection probability p as a function of infected devices n, and we observe that detection is practical even with few infected devices.

The goal of the collaborative detection system is to keep a constant, high detection probability at all times. This can be achieved with fewer devices

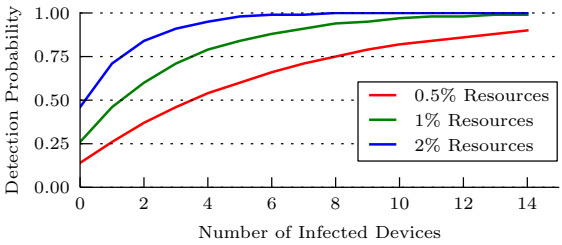


Figure 5.16: *The detection probability p as a function of infected devices n. We consider allocated system resources $s = \{0.5, 1, 2\}$% and assume $n_a = 5$. Detection is practical even with very low number of infected devices.*

sampling more often or more devices sampling less often. For example, the screenshot rate can be controlled based on the popularity (global install count) of the currently running, unknown app. The marketplace can send periodic updates on the popularity of each application installed on the device. If an app is present on many devices (e.g., 50 or more), the detection system can safely reduce the screenshot rate to save system resources without sacrificing detection probability. If an application is installed in only a small number of devices (e.g., less than 10), the system can increase the screenshot rate for better detection probability. Such adjustments can be done so that, in total, no more than the pre-allocated amount of system resources are spent for spoofing detection.

Our analysis has shown that collaborative detection provides an efficient way to detect spoofing attacks in the majority of practical spoofing scenarios.

5.8 Analysis

Collaborative detection. Extracting screenshots frequently and analyzing each of them can be expensive. However, if multiple devices take part in detection, we can reduce the overhead on every device without sacrificing detection probability. This can be achieved with fewer devices sampling more often or more devices sampling less often. For example, the screenshot rate can be controlled based on the popularity of the currently running, unknown app. If an app is present on many devices (e.g., 50 or more), the detection system can safely reduce the screenshot rate to save system resources without sacrificing detection probability. If an application is installed in only a small number of devices (e.g., less than 10), the system can increase the screenshot rate for better detection probability. Such

adjustments can be done so that, in total, no more than the pre-allocated amount of system resources are spent for spoofing detection.

We show that collaborative detection provides an efficient way to detect spoofing attacks in the majority of practical spoofing scenarios. For example, only 10 devices, each dedicating 1% of computation overhead, are needed to detect phishing attacks with a probability upwards of 95%.

Detection avoidance. The adversary can try to avoid runtime detection by leveraging the human perception property of averaging images that change frequently (e.g., quickly and repeatedly alternate between showing the first and second halves of the spoofing screen). The user would perceive the complete login screen, but any acquired screenshot would cover only half of the spoofing screen. Such attacks can be addressed by extracting screenshots frequently and averaging them out, prior to analysis.

While the adversary has an incentive to create spoofing screens that resemble the original login screen, the adversary is not limited to these modifications. To test how well our system is able to estimate deception rate for previously unseen visual modifications and spoofing samples that differ from the login screen model, further tests are needed. This limitation is analogous to the previously proposed similarity detection schemes that compare website to known phishing samples – the training data cannot cover all phishing sites.

Our current implementation has difficulties in decomposing screenshots with background noise, and consequently the adversary could try to avoid detection by constructing noisy spoofing screens. Developers could be encouraged to create clean login screen layouts for improved spoofing protection. While we did not experiment with noisy backgrounds, our study shows that the more the adversary deviates from the legitimate screen, the less likely the attack is to succeeded.

The goal of this work was to demonstrate a new spoofing detection approach, and we recommend that a deployed system be trained with more samples including (a) more visual modifications and (b) more apps.

False positives. Many mobile apps use single sign-on functionality from popular services, such as Facebook. An unknown application with a legitimate single sign-on screen matching to one of the reference values would be flagged by our detection system. Flagged applications should be manually verified and in such cases found benign.

5.9 Related Work

Spoofing detection systems. Static code analysis can be effective in detecting spoofing apps that leverage known attack vectors, such as ones that query running tasks and after that create a new activity [31]. Our approach is more agnostic to the attack implementation technique, but has a narrower focus: protection of login screens. We consider our work complementary to code analysis.

Many web phishing detection systems analyze a website DOM tree and compare its elements and structure to the reference site [12, 111, 154, 205, 208]. We assume an adversary that constructs spoofing apps in arbitrary ways (e.g., per pixel), and thus complicates structural code analysis.

Another approach is to consider the visual presentation of a spoofing application (or a website), and compare its similarity to a reference value [43, 72, 125]. Previous schemes typically derive a similarity score for a website and compare it to known malicious sites, while our metric determines how many users would confuse the application for another one.

Spoofing detection by users. Similar to web browsers, the mobile OS can be enhanced with security indicators. The OS can show the name of the running app in a dedicated part of the screen [31, 64, 162]. Such schemes require that parts of the mobile device screen are made unavailable to applications or need hardware changes to the device. A mobile app can also allow the user to configure a personalized security indicator (e.g., a personal image) that is shown by the app during each login [121].

Several studies, in the context of web sites, show that users tend to ignore the absence of security indicators [56, 160, 192]. A recent study shows that personalized security indicators can be more effective on mobile apps [121]. We are the first to study how likely the users are to notice spoofing attacks, where the malicious application resembles, but is not a perfect copy of, the legitimate application.

5.10 Conclusion

In this chapter, we have proposed a novel mobile app spoofing detection system that in collaborative fashion extracts screenshots periodically and analyzes their visual similarity with respect to protected login screens. We expressed similarity in terms of a new metric called deception rate that represents the fraction of users that would confuse the examined screen for one of the protected login screens. We conducted an extensive online user study and trained our detection system using its results. Our system

estimates deception rate with good accuracy (6% to 13% error) and low overhead (only 1%), and our system tells how likely the user is to fall for a potential attack. We consider this a powerful and interesting security property that no previous schemes provide. In addition to supporting a spoofing detection system, the results of our user study, on their own, provide insight into the perception and attentiveness of users during the login process.

System limitations. The main drawback of this approach is that the system runs on the user's device and performing complex visual similarity comparison is challenging, due to device constraints. We address this limitation in the following chapter.

Chapter 6

Cloud-based Spoofing Detection

6.1 Introduction

In the previous chapter, we proposed an on-device spoofing detection approach based on visual similarity. However, such an approach is *reactive* and can be considered as a last line of defense as malware is already installed on user devices. In this chapter, we propose a proactive approach, that detects malware *before* it reaches user devices.

Mobile application *visual impersonation* is the case where one application intentionally misrepresents itself in the eyes of the user. Such applications impersonate either the whole, or only a small part of the user interface (Figure 6.1). The most prominent example of whole UI impersonation is application repackaging; the process of republishing an app to the marketplace under a different author. It's a common occurrence [213] for an attacker to take a paid app and republish it to the marketplace for less than it's original price. In such cases, the repackaged application is stealing sales revenue from the original developers.

In the context of mobile malware, the attacker's goal is to distribute a malicious application to a wide user audience while minimizing the invested effort. Repackaging a popular app, and appending malicious code to it, has become a common malware deployment technique. Recent work [214] showed that 86% of analyzed malware samples were repackaged versions of legitimate apps. As users trust the familiar look and feel of their favorite

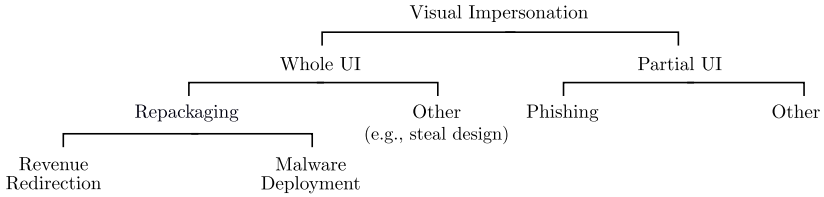


Figure 6.1: *Taxonomy of mobile app visual impersonation. Existing works primarily focused on detecting repackaging. Our goal is to detect all types of impersonation.*

apps, such a strategy tricks the users into believing that they are installing, and interacting with, a known app.

Application fingerprinting [21, 65, 79, 81, 86, 204, 207, 212, 215] is a common approach to detect repackaging. All such works compare fingerprints extracted from some feature of the application, such as their code or runtime behaviour. However, an adversary that has an incentive to avoid detection can *easily modify* all such features without affecting the appearance of the app, as seen by the user. For example, an attacker can obfuscate the app by adding dummy instructions to the application code. Code comparison approaches would fail because the new fingerprint would be significantly different, and we demonstrate that such detection avoidance is both effective and simple to implement.

Instead of impersonating the whole UI, malicious apps can also impersonate only a small part of the UI by, e.g., creating a fake login screen in order to phish login credentials. Phishing apps that target mobile banking have become a recurring threat, with serious incidents already reported [173, 176]. Prior works on repackaging detection and common malware detection [21, 75, 147, 184, 198] are ill-suited for detecting such phishing cases as the malicious apps share little resources with the original, they don't exhibit specific system call patterns, nor do they require any special permissions — they only draw to the device screen.

Due to these inherent limitations of previous detection techniques, we propose a conceptually different approach. Our goal is to design an impersonation detection system that is resistant to common detection evasion techniques (e.g., obfuscation) and that can be configured to efficiently detect different types of impersonation; from repackaging (whole UI) to phishing (partial UI). We observe that, for visual impersonation to succeed, and irrespective of possible modifications introduced to the app, *the ad-*

versary must keep the runtime appearance of the impersonation app close to the original. Our work in Chapter 5 showed that the more the visual appearance of a mobile app is changed, the more likely the user is to become alarmed. We propose a detection system that leverages this unavoidable property of impersonation.

Our system complements the one from Chapter 5 as well as existing fingerprint-based approaches, it runs on the marketplace and can analyze large amounts of Android apps using dynamic analysis and visual comparison, prior to their deployment onto the market. Our system runs the app inside an emulator for a specified time (e.g., 20 min), dynamically explores the mobile app user interface and extracts screenshots using GUI crawling techniques. If two apps have more than a threshold amount of screenshots in common (either exact or near matches), the apps are labeled as an instance of impersonation. In contrast to previous works, we do not base our detection on some easily modified feature of the app's resources, but rather on the final visual result of executing any app — the screenshot presented to the user. As a result, our system is robust towards introduced perturbations to either application resources, or to the way the UI is created — *as long as the application looks the same at runtime, our system will detect the impersonation*. No existing similar detection schemes offer this property.

To realize a system that is able to analyze a large number of apps, we had to overcome technical challenges. The existing GUI crawling tools [13, 130, 138] require application-specific knowledge or manual user input, and are therefore not applicable to automated, large-scale analysis. To address those challenges, we developed novel GUI crawling techniques that force the analyzed app to draw its user interface. Our system requires no user input, no application-specific knowledge, it supports the analysis of native applications, and thus enables automated analysis of apps at the scale of modern application marketplaces. Our system uses locality-sensitive hashing (LSH) [53] for efficient screenshots retrieval.

To evaluate our system, we dynamically analyzed over 150,000 applications downloaded from Google Play and other mobile app markets. Our system extracted approximately 4.3 million screenshots and found over 40,000 cases of impersonation; predominantly repacks (whole UI) but also apps that impersonate only a single registration screen (partial UI). These experiments demonstrate that impersonation detection through dynamic user interface extraction is effective and practical, even in the scale of large mobile application marketplaces.

6.2 Background on Android UIs

In this section we provide a concise primer on Android application user interfaces. Android apps with graphical user interfaces are implemented using activity components. A typical app has multiple activities, one for every separate functionality. For example, a messaging app could have activities for reading a message, writing a message, and browsing received messages. Each activity provides a window to which the app can draw its user interface, that usually fills the entire device screen, but it may also be smaller and float on top of other windows. The Android system maintains the activities of recently active apps in a stack. The window surface from the top activity is shown to the user or if the window of the top activity does not cover the entire screen, the user sees also the window beneath it. The user interface within a window is constructed using views, where each view is either a visible user interface element (e.g., a button or an image) or a set of subviews. The views are organized in a tree hierarchy that defines the layout of the user interface.

The recommended way to define user interface layouts in Android is using XML resource files, but Android apps can also construct view hierarchies (UI element trees) programmatically. Furthermore, Android apps can construct their user interfaces using OpenGL surfaces and WebViews. The OpenGL user interfaces are primarily used by games, while WebView is popular with many cross-platform apps. Inside a WebView, the user interface is constructed using common HTML. All of the methods stated above can be implemented in Java, as well as native code.

Every Android application contains a manifest file that defines the application's activities. One activity is defined as the application entry point, but the execution of the application can be started from other activities as well. When an activity is started, the Android framework automatically renders (inflates) the layout files associated with the activity.

6.3 Motivation and Case Study

Whole UI impersonation. The first type of impersonation we consider is whole UI impersonation (Figure 6.1). Since a typical repackaged app shares majority of its code and runtime behaviour with the original application, fingerprinting can be an effective way to detect such impersonation. Known techniques leverage static application features, such as code structure [65], imported packages [215] and dynamic features [21, 75, 147, 184, 198], such as system call [108] or network traffic patterns [163]. Using fingerprinting, large number of repacks have been detected from several

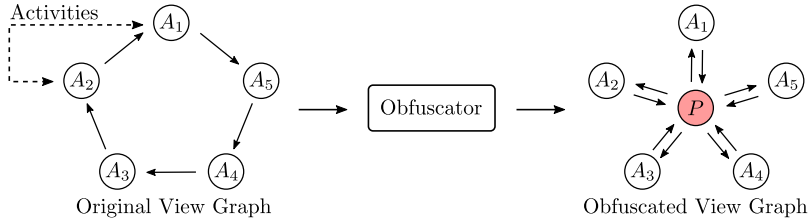


Figure 6.2: *Obfuscation example. We developed a small obfuscation tool that modifies the activity transition graphs of an app without affecting user experience. The obfuscation introduces additional proxy activities that mediate UI screen transitions. In the original app, the user transitions (e.g., by clicking a button)* $A_1 \rightarrow A_5$*. Afterwards, the transition becomes* $A_1 \rightarrow P \rightarrow A_5$*. The transition occurs so fast that the user does not notice it.*

Android application marketplaces [41]. The inherent limitation of fingerprinting approaches is that the adversary can easily modify the above listed (or any similar) features without affecting the runtime appearance of the application towards the user. In this section we demonstrate that such modifications are both easy to implement and effective in avoiding detection.

Case study. We present a case study on MassVet [41], a recent static analysis system that has demonstrated good results in detecting repackaging. MassVet extracts two types of static fingerprints from a mobile application: *m-cores* define application code flow graph and sequence of instructions and *v-cores* express UI state transitions from one activity to another. The rationale behind these fingerprints is that repackaged apps typically share majority of their functionality, as well as user interface structure.

We built a simple obfuscation tool (less than 200 lines of code) that modifies both extracted features (m-cores and v-cores). Our tool adds blocks of unused code, as well as proxy activities that modify activity transitions. Our modifications have no visible effect on the user experience (see Figure 6.2). To evaluate the tool, we took 1259 apps from the Android Malware Genome project [1]. We ran those apps through the MassVet tool online interface[1] and 559 applications were identified as potentially malicious repackaged apps. We obfuscated all of these apps using our tool and rerun them on the MassVet system — 92% of the obfuscated apps were no longer detected. We argue that similar limitations apply to any

[1] `http://www.appomicsec.com/`

fingerprinting solution and adoption of obfuscation introduces little to no cost to repackaging adversaries, as it merely implies running an automated tool. We demonstrated only one possible approach, and such obfuscation tools can be implemented for all prior fingerprinting approaches (both static and dynamic), in a similar manner and level of complexity.

Partial UI impersonation. The second type of visual impersonation are apps that only impersonate a small portion of the UI. Application phishing is an example of impersonation, where a malicious application constructs a single screen that visually resembles one of a another app, but otherwise shares no similarities (e.g., no shared code or resources) with the original app. For example, a malicious game asks the user to perform an in-app purchase, but instead of starting the legitimate banking app, the game presents a phishing screen that mimics the login screen of the bank. Distinguishing the phishing screen from the genuine login screen is difficult for the user, as modern mobile platforms do not enable the user to identify the application currently drawing on the screen.

Fingerprinting is not effective against such impersonation apps, as the malicious app does not share large parts of resources (e.g., code). Furthermore, traditional mobile malware detection schemes that, e.g., examine API call patterns [31] and permissions [27] are also ill-suited for the detection of phishing. Such applications differ from regular malware as they often require no special permissions, nor necessarily perform suspicious actions. They only perform a single operation — *drawing on the device screen*.

Adversarial model. We consider a strong attacker that can implement impersonation in various ways. The attacker can create the UI by, e.g., using standard OS libraries, implement it in a custom manner, or show static images of the interface. On Android, the attacker could implement the app in Java, native code, or as a web app. For a more thorough introduction of the various ways of creating user interfaces in Android, we refer the reader to Section 6.2. On top of that, the adversary can modify the application code or resource files in arbitrary ways (e.g., obfuscation). Such an adversary is both realistic and practical. The attacker can freely create the impersonated screens by any means allowed by the underlying Android system. Running an (off-the-shelf or custom) obfuscation tool comes at little to no cost to the adversary.

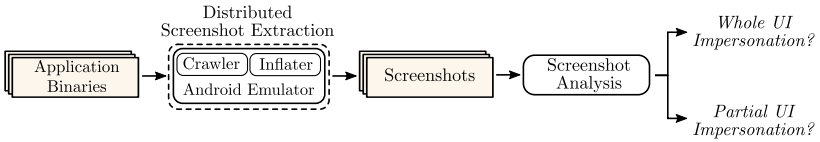


Figure 6.3: *An overview of the impersonation detection system that works in two phase. First, we extract user interfaces from a large number of applications. Then we analyze the extracted screenshots to detect repackaging and impersonation.*

6.4 Visual Impersonation Detection System

As demonstrated in the previous section, the adversary has significant *implementation freedom* in performing an impersonation attack and, at the same time, the adversary has a clear incentive to keep the visual appearance of the app close to the original. Our work in Chapter 5 has shown that the more the adversary deviates from the appearance of the original mobile application user interfaces, the more likely the user is to become alarmed. We say that the adversary has limited *visual freedom* and our solution leverages this property.

Our goal is to develop a robust visual impersonation detection mechanism that is based on visual similarity — a feature the attacker cannot modify without affecting the success of the attack. More precisely, the system should: (i) detect both whole (e.g., repackaging) and partial UI impersonation (e.g., phishing), (ii) be robust towards the used impersonation implementation type and applied detection avoidance method (e.g., obfuscation), (iii) analyze large numbers of apps in an automated manner (e.g., on the scale of modern app stores).

Figure 6.3 shows an overview of our system that works in two main phases. In the first phase, the system extracts user interfaces from a large number of mobile apps in a distributed and fully autonomous manner. The system takes as input only the application binary, and requires no application-specific knowledge or manual user input. We run each analyzed app in an emulated Android environment, and explore its user interface through crawling. An attacker could develop a malicious app that detects emulated environments. However our system can be executed on real hardware as well (Section 6.8). In a best-effort manner, we extract as many screenshot as possible within a specified time limit. Full exploration coverage is difficult to achieve [24, 181], and as our results in Section 6.5

show, not necessary for effective impersonation detection. During crawling, our system also automatically identifies reference screens (e.g., ones with login and registration functionality) that benefit from our protection.

In the second phase, the system examines the extracted screenshots to find cases of impersonation. To detect whole UI impersonation, our system finds applications that share the majority of their screenshots with a queried app. To detect partial UI impersonation, our system finds applications that share a similar (e.g., login or user registration) screen with the reference app, but have otherwise different screenshots.

Our system can be deployed on the marketplace and used to detect impersonation, e.g., upon submitting the application to the market. We emphasize that our system can be combined with existing approaches to enhance impersonation detection. For example, only applications that are considered as benign by a fast fingerprint-based approach could be submitted to our, more costly analysis.

6.4.1 Design Challenges

To realize the system outlined above, we must overcome a number of technical challenges. First, the existing GUI crawling approaches were designed for development and testing, and a common assumption is that the test designers have access to application-specific knowledge, such as source code, software specifications or valid login credentials. Login screens are a prominent example of how crucial application-specific knowledge is in GUI exploration, as such tools need to know a valid username and password to explore beyond the login screens of apps. Another example are games where, in order to reach a certain GUI state, the game needs to be won. In such cases, the exploration tool needs to be instructed how to win the game. Previous crawling tools address these issues of reachability limitations using application-specific exploration rules [24, 199] and predefined crawling actions [201]. As our system is designed to analyze a large number of mobile apps, similar strategies that require app-specific configuration are not possible. In Section 6.4.2 we describe a mobile app crawler that works fully autonomously, and in Section 6.4.3 we describe new user interface exploration techniques that increase its coverage.

Second, dynamic code analysis is significantly slower than static fingerprinting. In Section 6.4.2 we describe a distributed analysis architecture that enables us to analyze applications in a fully scalable manner. And third, many repackaged apps and known phishing malware samples [176] contain minor visual differences to their target apps. Our system must efficiently find screenshots that are exact or near matches, from a large set of screen-

shots. In Section 6.4.4 we describe a system that uses locality-sensitive hashing [53] for efficient screenshot analysis.

6.4.2 Automated Crawling

We designed and implemented a UI crawler as part of the Android core running inside an emulator (Figure 6.3). The crawler uses the following basic strategy. For each new activity, and every time the view hierarchy (tree of UI elements) of the current activity changes, our crawler takes a screenshot. The crawler continues exploration in a depth-first manner [24], as long as there are clickable elements (views) in the user interface. To support autonomous user interface exploration, our crawler must determine which views are clickable without prior knowledge. We implemented the crawler as a part of the Android core, which gives it full access to the state of the analyzed app, and we examine the current state (i.e., traverse the view tree) to identify clickable elements. We observed that in many apps, activities alter their view hierarchy shortly after their creation. For example, an activity might offload a lengthy initialization process to a background thread, show a temporary UI layout first, and the final one later. To capture such changes, our crawler waits a short time period after every transition.

To increase the robustness of crawling, we made an additional modification to the Android core. If the crawled application creates an Intent that triggers another app to start, we immediately terminate it, and resume the execution of the analyzed app. In practice this approach turned out to be an efficient way to continue automated user interface exploration.

Reference screen identification. To enable partial UI impersonation detection, our system automatically identifies screens that benefit from impersonation protection. We have tailored our implementation to identify screens that contain login or registration functionality. While crawling an app, we traverse the view hierarchy tree of each screen and consider the screen a possible login or user registration screen, when the hierarchy contains at least one password field, one editable text field, and one clickable element or meets other heuristics, such as the name of the activity contain word “login” or “register”. If such a screen is found, we save it as a *reference screen* for partial UI impersonation detection (Section 6.4.4). Reference screen identification is intended for benign apps that have no incentive to hide their UI structure, but we repeat the process for all crawled apps, since we do not know which apps are benign.

Distributed architecture. We built a distributed analysis architecture that leverages cloud platforms for analysis of multiple apps. Our architecture consists of a centralized server and an arbitrary number of analysis instances. The server has a database of apps, orchestrates the distributed analysis process, and collects the extracted user interfaces. Each analysis instance is a virtual machine that contains our dynamic analysis tools.

6.4.3 Coverage Improvements

In this section we describe techniques we implemented to increase the coverage of our user interface exploration.

Out-of-order execution. The crawler starts from the entry-point activity defined in the manifest file. For some apps only a small part of the user interface is reachable from the entry point without prior application-specific knowledge (e.g., a login screen requires valid credentials to proceed). To improve our crawling coverage, we additionally force the application to start all its activities out of order. This is a best-effort approach, as starting the execution from an arbitrary activity may crash the app without correct Intent or application state.

Layout file inflation. We implemented a tool that automatically renders (inflates) mobile app user interfaces based on XML layout files and web resources. As many apps customize the XML layouts in code, the final visual appearance cannot be extracted from the resource file alone. We perform resource file inflation from the context of the analyzed app which ensures that any possible customization will be applied to UI elements defined in the resource file. We implemented a dedicated enforcer activity and force each app to load it at startup. This activity iterates through the app's layout files, renders them one by one and takes a screenshot. We noticed that increasingly many mobile apps build their user interface using web technologies (e.g., HTML5). To improve the coverage of such apps, we perform similar inflation for all web resources. Our enforcer activity loads all local web resources of the application one by one.

Layout file inflation is conceptually different from, e.g., extracting fingerprints from resources. The attacker can modify layout files without affecting our analysis, as we take a screenshot of the final rendered layout.

User interface decomposition. Our crawling approach relies on the assumption that we can determine all clickable UI elements by analyzing the view hierarchy of the current activity. While this assumption holds for

many apps, there are cases where clickable elements cannot be identified from the view tree, and therefore our crawler cannot proceed. For instance, mobile apps that implement their user interfaces in OpenGL (mostly games) and malicious apps that intentionally attempt to hide parts of their user interface from crawling.

We integrated a user interface decomposition outlined in Chapter 5 to our crawler to improve its coverage and attack resistance. Our experiments (Section 6.5) show that by using this extension we are able to crawl a number of apps whose UIs we would not be able to crawl otherwise.

6.4.4 Screenshot Analysis

Our initial experiments showed that many screenshots extracted from repackaged apps have minor visual differences in them. Some of the observed differences are caused by the application (e.g., different version, language or color scheme) while others are artifacts of our exploration approach (e.g., a blinking cursor visible in one screenshot but not in the other). Also the phishing screens seen in known malware samples contain minor visual differences to their target apps [176].

Our system handles visual differences in the extracted screenshots using *perceptual hashing*. The goal of a perceptual hash is to provide a compact representation of an image that maintains its main visual characteristics. In contrast to a cryptographic hash function, a small modification in the image produces a hash value that is close to the hash of the original image. Perceptual hashing algorithms reduce the size and the color range of an image, average color values and reduce their frequency. The perceptual hash algorithm we use [35] produces 256-bit hashes and the visual similarity of two images can be calculated as the Hamming distance between two hashes. To enable analysis of large number of screenshots, we leverage *locality-sensitive hashing* [53] (LSH). LSH algorithms are used to perform fast approximate nearest neighbour queries on large amounts of data.

Using these two techniques (perceptual hashing and LSH), we built a screenshot analysis system (Figure 6.4). The system consists of three operations: (1) indexing, as well as detection for both (2) whole and (3) partial UI impersonation attacks.

Indexing. In the indexing phase, a perceptual hash is created for each extracted screenshot and all hashes are fed to our LSH implementation. We use bit sampling for reducing the dimensionality of input items, as our perceptual hashing algorithm is based on the Hamming distance. We set the LSH parameters through manual experimentation (Section 6.5). To

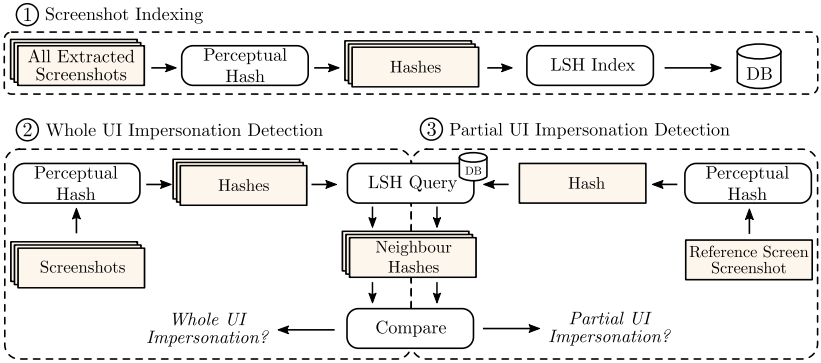


Figure 6.4: *Screenshot analysis system for impersonation detection. All extracted screenshots are indexed to a database using locality-sensitive hashing (LSH). To find impersonation applications from a large dataset, we first find nearest neighbor screenshots using the LSH database, and then perform an more expensive pairwise comparison.*

reduce false positives, we manually created a list of screenshots belonging to shared libraries (e.g., ad or game frameworks), and we removed all such screenshots from the database.

Whole UI impersonation detection. To detect impersonation, we use the intuitive metric of *containment* (C), i.e., how many screenshots of one app are included in the screenshot set of another app. To find whole UI impersonation apps, our system takes as input a queried app, and finds all apps whose user interface has high similarity to it. We note that, while our system is able to detect application pairs with high user interface similarity, it cannot automatically determine which of the apps in the relationship (if any) is the original one.

Our system analyzes one app at a time, and the first step in whole UI impersonation detection is to obtain the set of all screenshots (Q) of the current queried app. For every screenshot, we hash it and query the indexed LSH database. For each query, LSH returns a set of nearest neighbour screenshot hashes, their app identifiers and distances to the queried hash. For returning the nearest neighbours, we use the cutoff hamming distance $d = 10$, as explained in Section 6.5.

We sort this dataset based on the application identifiers to construct a list of candidate applications, where P_i is the set of their screenshots. For each candidate app we find the best screenshot match to the queried app. As a result, we get a set of matching screenshots for each candidate app. We consider the candidate app as a potential impersonation app when (1) the ratio (containment C) between the number of matched app and reference app screenshots is larger than a threshold T_w, and (2) the number of considered screenshots meets a minimum threshold T_s. Without loss of generality, we assume $|P_i| \leq |Q|$.

$$C = \frac{|P_i \cap Q|}{|P_i|} \geq T_w \tag{6.1}$$

$$|P_i| \geq T_s \tag{6.2}$$

In Section 6.5 we describe the experiments we used to set these thresholds (T_w, T_s). To find all potential repacks from a large dataset of applications, the same procedure is repeated for each application.

Partial UI impersonation detection. For partial UI impersonation, we no longer require a significant visual similarity between the two apps (target and malware). Only specific screens, such as login or registration screens, must be visually similar to perform a convincing impersonation attack of this kind. To scan our dataset for such applications, we adjusted the search criteria accordingly. Given our set of potential reference screens (Section 6.4.2) extracted during our dynamic analysis phase, we target applications that contain the same or a very similar screen but otherwise do not share a significant visual similarity in other aspects of the application. We only consider applications to be of interest if their containment with the queried application is less than a threshold ($C \leq T_p$), as long as the app contains the queried login or registration screen.

6.5 Evaluation

In this section we evaluate the detection system. For evaluation we downloaded 158,449 apps from Google Play and other Android application repositories (see Table 6.1). From Google Play we downloaded approximately 250 most popular apps per category. Our rationale was that popular apps would be likely impersonation targets. We also included several third-party markets, as repacks are often distributed via such stores [41].

Marketplace	apps
play.google.com (US)	14,323
coolapk.com (CN)	5666
m.163.com (CN)	24,069
1mobile.com (CN)	24,173
mumayi.com (CN)	29,990
anzhi.com (CN)	36,202
slideme.org (US)	19,730
android.d.cn (CN)	4635
Total	**158,449**

Table 6.1: *Application dataset. The list of downloaded and analyzed applications (per application market).*

6.5.1 User Interface Extraction

Analysis time. We deployed our system on the Google Cloud Compute platform. The analysis time of a single application varies significantly, as apps have user interfaces of different size and complexity. While the majority of apps have less than 20 activities, few large apps have up to 100 activities. Furthermore, some applications (e.g., games) may dynamically create a large number of new user interface states and in such cases the dynamic user interface exploration is an open ended process. To address such cases, in our tests we set the maximum analysis time to generous 45 minutes. On the average, the analysis of one mobile app took only 7 minutes and we plot the distribution of analysis time in Figure 6.5. Extracting screenshots is the most time-consuming part of our system. Once the screenshots are extracted, querying LSH and deciding if any impersonation exists is fast (few seconds per app).

At the peak of our analysis we used 1000 computing instances on the Google Cloud Compute platform and the entire analysis of over 150,000 apps took 36 hours. At the same rate and similar computing platform, the entire Google Play market (1.6 million apps) could be analyzed in approximately two weeks. We consider this a feasible one-time investment and the overall analysis time could be further reduced by limiting the maximum analysis time further (e.g., to 20 minutes as discussed below).

Analysis coverage. From the successfully analyzed 139,656 apps we extracted over 4.3 million screenshots after filtering out duplicates from the same application and low-entropy screenshots, e.g., single-color back-

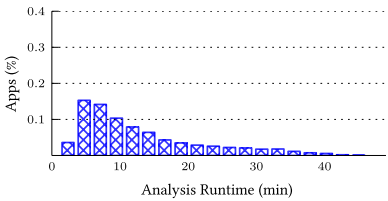


Figure 6.5: *The distribution of analysis time.*

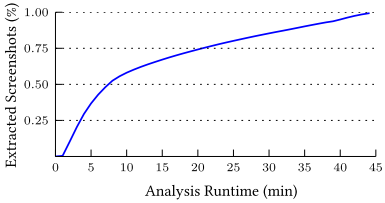


Figure 6.6: *Average number of screenshots extracted from an app, as a function of time.*

grounds that are not useful in impersonation detection. The majority of applications produced less than 50 screenshots, but we were able to extract up to 150 screenshots from some apps. Figure 6.6 plots the percentage of extracted screenshots as a function of analysis time. We extract approximately 75% of all the screenshots during the first 20 minutes of analysis. The steeper curve during the first 7 minutes of the analysis is due to the fact that we run the inflater tool first and after that we start our crawler. The crawler tool extracted 58% of the screenshots and the inflater contributed additional 42%. Majority (97%) of the extracted screenshots come from user interfaces implemented using standard Android user interface elements and a small share originates from user interfaces implemented using web techniques or as an OpenGL surface. Figure 6.10 summarizes the user interface extraction results.

We use *activity coverage* as a metric to compare the coverage of our crawler to previous solutions. Activity coverage is defined as the number of explored activities with respect to the total number of activities in an app [24]. While activity coverage does not account for all possible user interface implementation options, it gives a good indication of the extraction coverage when analyzing large number of apps. Our tool achieves 65% activity coverage, and is comparable to previous solutions (e.g., 59% in [24]). However, previous crawling tools that achieve similar coverage require application-specific configuration or manual interaction.

We separately tested our user interface decomposition extension on 200 apps (mostly games) that implement their UI using OpenGL surface. Without the decomposition extension, our crawler was only able to extract a single screenshot from each of the tested apps (the entry activity). With decomposition, we were able to extract several screenshots (e.g., 20) from 30% of the tested apps. This experiment demonstrates that there is a class

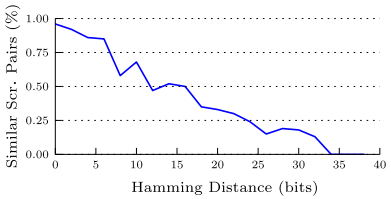


Figure 6.7: *Manual evaluation of hamming distances of screenshot hashes.*

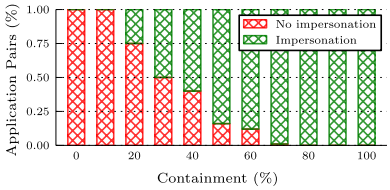


Figure 6.8: *Manual evaluation of false positives and false negatives.*

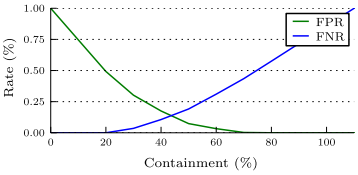


Figure 6.9: *False positive and false negative rates. Equal-error rate is at cont. 45%.*

Number of successfully analyzed apps	**139,656**
Number of extracted screenshots	**4,302,413**
Extracted by the crawler	57.89%
Extracted by the inflater	42.11%
Originated from widget-based UI	96.95%
Originated from a HTML asset	2.17%
Originated from an OpenGL surface	0.88%

Figure 6.10: *User interface extraction results.*

of mobile apps whose user interfaces can be crawled with better coverage using decomposition.

6.5.2 Screenshot Comparison

Perceptual hash design. We investigated two different perceptial hashing techniques, based on image moments [177] and DCT [35] (pHash). Preliminary analysis of the results from both hashing techniques revealed that the image moments based approach is not suitable for the hashing and comparing of user interface screenshots, as both very similar as well as very dissimilar pairs of screenshots resulted in almost equivalent distances. The *pHash* [202] based approach yielded promising results with a good correlation between visual similarity of user interface screenshots and the hamming distance between their respective hashes. Screenshots of user interfaces are quite different from typical natural images: clearly defined boundaries from UI elements, few color gradients, and a significant amount of text. To account for these characteristics and improve the performance of the hashing algorithm, we performed additional transformations (e.g., dilation) to the target images as well as the final hash computation.

To find a good cutoff hamming distance threshold below which we consider images to be visually similar, we randomly selected pairs of images and computed their distance until we had 100 pairs of screenshots for the first 40 distances. We then manually investigated each distance bucket and counted the number of screenshot pairs we considered to be visually similar. The result are shown in Figure 6.7, and we concluded that a distance $d = 10$ is a reasonable threshold.

Containment threshold. Similar to the distance threshold evaluation, we randomly selected pairs of apps from our dataset and computed their containments, creating 10 buckets for possible containment values between $[0, 100]$ percent, until we had 100 unique pairs of apps for all containment values. We then manually examined those apps and counted the number of application pairs which we consider to be whole UI impersonations, as shown in Figure 6.8. The false negative and false positive rates are shown in Figure 6.9, yielding $T_w = 0.45$ containment to be a reasonable threshold for whole UI impersonation detection, above which applications within a pair are considered impersonation cases of each other. To verify that our manually derived false negatives rates were representative, we performed an additional check outlined below. To detect partial UI impersonation, in our experiments, we found that setting the containment threshold to $T_p = 0.10$ gave good results.

Containment threshold verification. To further verify our manual containment threshold evaluation, we created a large ground truth of application pairs known to be trivial repacks of each other. We define a trivial repacks as any pair of applications where both applications have at least 90% identical resource files as well as a code similarity (computed using Androguard) of 90% or higher. For such pairs, we can be confident that the apps are indeed repacks. Over the course of several days, we compiled a list of 200,000 pairs to serve as a ground truth of known repacks. We then ran our system against every pair in this list, querying either app of the pair. If the system did not find the corresponding app, or if their containment is below the threshold value of 45%, we consider this a false negative. We repeated this exercise for different threshold values and compared the results to the expected false negative rates, confirming that the pairs considered in our manual verification are indeed representative for our large data-set.

6.6 Detection Results

To demonstrate that our system is able to catch impersonation attacks, we analyzed each application in our dataset and we report the following results.

Whole UI impersonation. Using perceptual hash distance $d = 10$ and containment threshold $T_w = 0.45$, our system detected 43,904 impersonating apps out of 137,506 successfully analyzed apps. Such a high number of impersonation instances is due to the fact that a large part of our apps are from third-party markets. From the set of detected apps, our system does not automatically detect which apps are originals and which are impersonators.

At $T_w = 0.45$ our system has an estimated 15% false negatives. We remind the reader that these false negatives refer to missed *application pairs*, and not missed repacks. To illustrate, let us consider a simple example where our system is deployed on a market, and 3 apps are submitted (the original Facebook app, and 2 apps that impersonate it). The first app will be added and no impersonation can be detected. On the second app, our system has a 15% to miss the relation between the queried app and the one already in the store. However, the third app has approximately only $0.15^2 = 0.02$ chance of missing both relations and not being identified as impersonation with regards to the other two apps in the store. During our experiments, we found instances of clusters containing up to 200 apps, all repackaging the same original app.

Similarly, at $T_w = 0.45$ our system has also 15% false positives, which is arguably high. However, due to the fact that apps are repacked over and over again, on a deployed system we can set the containment threshold to be higher (e.g., $T_w = 0.60$). At that value, our system has only 3% false positives, and 31% false negatives. However, if the app is repacked a modest number of 5 times, the chances of missing all impersonation relationships becomes less than half a percent $0.31^5 = 0.002$.

In the above examples, we assumed that the analysis of each app is an independent event. In reality, this may not be the case. For example, if one impersonation app is missed, a closely related impersonation app may be missed with higher probability (i.e., the events are not independent). However, the more apps impersonate an original app, the higher the chances of our system catching it.

Partial UI impersonation. Using the metric described in Section 6.4.4, and $T_p = 0.10$, we found approximately 1,000 application pairs that satisfy the query. We randomly selected and manually inspected 100 pairs of apps

to understand their relationships. In most cases, we found repackaged apps with a large number of additional advertising libraries attached.

Among these results, we also found an interesting case of a highly suspicious impersonation application. In this sample, the impersonation target is a dating app. The registration screen of the impersonation app appears visually identical to original. However, manually inspecting the code reveals that a new payment URL has been added to the application, no longer pointing to the dating website, but instead to a different IP. We uploaded the sample to virustotal.com to confirm our suspicions, and the majority of scanners indicated maliciousness. The code similarity between the original and impersonating apps (according to Androguard) is only 22%, largely due to added advertising libraries.

Our similarity metric allows us to find specialized kinds of impersonation attacks that deviate from the more primitive repackaging cases. Interesting user interfaces with certain characteristics (e.g. login behaviour) can be queried from a large data-set of analysed applications to find various kinds of impersonation, drastically reducing the necessary manual verification done by humans.

6.7 Security Analysis

Detection avoidance. Our user interface extraction system executes applications in an emulator environment. A repackaging adversary could try to fingerprint the execution environment and alter the behavior of the application accordingly (e.g., terminate execution if it detects emulator). The obvious countermeasure is to perform the user interface exploration on real devices. While the analysis of large number of apps would require a pool of many devices (potentially hard to organize), user interface exploration would be faster on real devices [136] compared to an emulator. Our user interface extraction system could be easily ported from the emulator environment to real Android devices.

A phishing adversary could construct the phishing screen in a way that complicates its extraction. To avoid the inflater, the adversary can implement the phishing screens without any resource files. To complicate extraction by crawling, the adversary could try to hide the phishing screen behind an OpenGL surface that is hard to decompose and therefore explore or make the appearance of the phishing screen conditional to an external event or state that does not necessarily manifest itself during the application analysis. Many such detection avoidance techniques reduce the likelihood that the phishing screen is actually seen by the user (a necessary

precondition of any phishing attack). While our solution does not make impersonation impossible, it raises the bar for attack implementation and reduces the chances that users fall for the attack.

6.8 Discussion

Improvements. Our crawler could be extended to create randomized events and common touch screen gestures (e.g., swipes) for improved coverage. Our current out-of-order execution could be improved as well. Currently, startup of many activities fails due to missing arguments or incompatible application state. One could try to infer these from the application code through static analysis. Static code analysis could be also used to identify sections of application code that perform any drawing on the screen and the application could be attempted to force the execution of this code segment, e.g., leveraging symbolic execution. We consider such out-of-order execution a challenging problem.

Deployment. The primary deployment model we consider is one where a marketplace, mobile platform provider, anti-virus vendor or a similar entity wants to examine a large number of mobile apps to detect impersonation in the Android ecosystem. As shown in Section 6.5, the analysis of large number of apps requires significant resources, but is feasible for the types of entities we consider. Once the initial, one-time investment is done, detection for new applications is inexpensive. User interface extraction for a single application takes on the average 7 minutes and finding matching repacks or phishing apps can be done in the matter of seconds. As a comparison, the Google Bouncer system dynamically analyzes each uploaded app for approximately 5 minutes [141]. Our system streams screenshots to the central analysis server as they are extracted. The system can therefore stop further analysis if a decision can be made.

Post-detection actions. Once an application has been identified as a potential impersonation app, it should be examined further. Ad revenue stealing repacks could be confirmed by comparing the advertisements libraries and their configurations in two apps with matching user interfaces; sales revenue stealing repacks could be confirmed by comparing the publisher identities; repacks with malicious payload could be detected using analysis of API calls [20], comparison of code differences [41], known malware fingerprints or manual analysis; and phishing apps could be confirmed by examining the network address where passwords are sent. Many post-

detection actions could be automated, but implementation of these tools is out of scope for this work.

6.9 Related Work

Repackaging detection. The previous work on repackaging detection is mostly based on static fingerprinting. EagleDroid [175] extracts fingerprints from Android application layout files. ViewDroid [204] and MassVet [41] statically analyze the UI code to extract a graph that expresses the user interface state and transitions. The rationale behind these works is that while application's code can be easily obfuscated, the user interface structure must remain largely unaffected. We have showed that detection based on static fingerprinting can be easily avoided (Section 6.3) and our solution provides a more robust means of repackaging detection, at the cost of increased analysis time.

Phishing detection. The existing application phishing detection systems attempt to identify API call sequences that enable specific phishing attacks vectors (e.g., activation from the background when the target application is started [31]). While such schemes can be efficient to detect certain, known attacks, other attacks require no specific API calls. Our solution applies to many types of phishing attacks, also ones that leverages previously undiscovered attack vectors.

User interface exploration. The previous work on mobile application user interface exploration focuses on maximizing GUI coverage and providing means to reason about the sequence of actions required to reach a certain application state for testing purposes [13, 24, 101, 199]. These approaches require instrumentation of the analysis environment with application-specific knowledge, customized companion applications, or even source code modifications to the target application. Our crawler works autonomously and achieves similar (or better) coverage.

6.10 Conclusion

In this chapter, we have proposed and demonstrated a novel approach for mobile app impersonation detection. Our system extracts user interfaces from mobile apps and finds applications with similar user interfaces. Using the system we found thousands of impersonation apps. In contrast to previous fingerprinting systems, our approach provides improved resistance

to detection avoidance, such as obfuscation, as the detection relies on the final visual appearance of the examined application, as seen by the user. Another benefit of the system is that it can be fine-tuned for specialized cases, like the detection of phishing apps. Finally, the novel user interface exploration techniques that we have developed as part of this chapter, may have numerous other applications besides impersonation detection.

System limitations. The main drawback of our approach is the significantly increased analysis time, when compared to static analysis. However, our experiments show that such dynamic impersonation detection at the scale of large application stores is practical.

Chapter 7

Closing Remarks

In this final chapter, we summarize our findings, state lessons learned, as well as opinions on both the problem of user interface security, and possible future directions for the field.

7.1 Summary

We began this thesis by motivating the problem of user interface security. We summarized related work in Chapter 2, and pointed out their benefits as well as drawbacks. We showed that there is room for improving the state-of-the-art both in terms of known user interface attacks, as well as countermeasures.

In the first part of the thesis, we demonstrated two novel attacks that overcome the limitations of prior work. In Chapter 3, we demonstrated an input inference attack for Android smartphones that enables precise and continuous inference of user clicks, and that can be implemented without any special permissions. We achieved that goal by utilizing a previously unknown side-channel based on hover technology. Through our attack, we illustrated how the introduction of new input methods can have unexpected and far-reaching consequences on the security of the overall system. In Chapter 4, we proposed a new class of accurate and stealthy command injection attacks. We showed how our attack approach can infer system state by only observing the constant stream of user input, and can launch precise and damaging UI attacks. Through a proof-of-concept attack device, we demonstrated how such attacks can operate on low-end hardware,

and are realistic threats to a wide range of devices (from regular desktop systems, to dedicated terminals).

In the second part of the thesis we focused on preventing a widespread and efficient user interface attack; namely mobile application phishing. Contrary to prior works, we approached the problem by using visual similarity, as perceived by the user. In Chapter 5, we proposed an on-device spoofing detection system that acquires and compares screenshots at runtime. We performed user studies and we gained insight into how users perceive visual modifications in login screens. We have shown that our approach is more resilient to obfuscation attacks, as the more the attacker changes the visual presentation of the phishing attack, the more the attacker risks users detecting it. In Chapter 6, we explored our visual similarity approach further. We proposed a system tailored for detecting spoofing attacks at large scales (e.g., at the level of a whole marketplace). We implemented the system, and demonstrated that it can both efficiently and accurately detect spoofing attacks in a scalable manner.

7.1.1 Future Work

In this thesis, we considered widespread attack scenarios where an adversary could either (1) run a malicious application on the system, or could (2) through physical access exchange existing or introduce new peripherals to the system. Others considered the problem when the OS is compromised (e.g., trusted paths). However, alternative types of user interface attacks are also possible.

Clark and Blumenthal [45] wrote an insightful paper on what the concept of end-to-end security means on modern computer systems. Although their work was written in the context of computer networks, their reasoning also applies to the field of user interface security. The authors describe their "*ultimate insult*", namely the observation that if a user does not trust their own end-point (e.g., their own device), then having a secure end-to-end channel between their local end-point and a another remote end-point (e.g., server) is rendered meaningless.

If as end-points we consider not the devices themselves, but the input and output peripherals through which the user is interacting with the system, we can interpret the observation in the following way. If the users do not trust their input and output peripherals then, irrespective of the applied protection mechanisms, user interface attacks are possible.

To illustrate the idea, consider the following setting. The user interacts with a computer system that enforces secure boot and enables only signed and manually vetted applications to run. Therefore, the system prevents any

kind of malware from running on the device (unless the attacker acquires the signing key). Furthermore, the system enforces the use of trusted input and output peripherals that, e.g., share keys with the system and can therefore ensure that user input is both authenticated and confidential while in transit, which would prevent command injection or man-in-the-middle attacks. We can then ask the following question: "*Is such a system secure against user interface attacks?*".

The attacker can still perform various physical attacks. For example, the attacker could violate the confidentiality of input and output through shoulder-surfing. The attacker could also physically apply a thin layer over the peripherals (e.g., similar to a skimming device) and thereby compromise both confidentiality and integrity. We can conclude that the adversary can also perform attacks against the channel between the user and the peripherals, and that user interface attacks are therefore possible as long as the user interacts with peripherals in an environment the attacker can influence.

Securing the user-to-peripheral channel. A possible approach to reduce the impact of such attacks is to enable the user constant control over the peripherals. For example, if users carry their own optical head-mounted displays such as Google Glass [8] or future direct neural user interfaces, it is significantly harder for the attacker to compromise such devices than, e.g., devices the user leaves unattended for extended periods of time (e.g., input peripherals). However, securing the channel between the user and the peripherals remains an open problem.

Preventing adaptive attacks. In Chapter 4, we presented a new kind of physical attack where the malicious hardware device injects commands through the legitimate input channel that the user would normally use. Preventing such class of attacks is an open question, and possible approaches could include biometric solutions (e.g., fingerprinting the typing pattern, or other forms of biometry) that would enable the target system to distinguish if input comes from the legitimate user.

Modeling human perception. The overarching idea in the second part of this thesis (Chapter 5 and Chapter 6) was that users are bad at detecting security indicators, and are overall susceptible to phishing attacks, and that spoofing detection systems should therefore be automated instead of relying on the user's attention. To that end, we proposed two automated systems that detect mobile application spoofing attacks. Both of our systems are based on visual similarity, and knowledge extracted (modeled)

from users. However, our user studies were but an initial step towards that direction, as the better we model user perception, the higher quality inferences about detecting UI attacks we can make. A possible future work is to explore other kinds of visual modifications that we did not consider in this thesis (e.g., horizontally or vertically displacing UI elements), and include those in the model as well.

Assisting automated approaches. We observed that some security-critical user interfaces (e.g., login screens) are visually much more complex than others. For example, some login screens featured background natural images, while others did not. For a spoofing detection system based on visual similarity, visual complexity is an obstacle, and the simpler the login screen design, the better our approaches would work in detecting attacks.

User interfaces have strict requirements. They have to be intuitive, and easy to use, which often requires giving UI designers full freedom to explore and innovate. Currently, UI designers have no visual constrains when it comes to designing login screens. However, by imposing moderate restrictions on how designers can create such security-critical UIs (e.g., only a certain number of elements, positioned at pre-determined positions, etc.), automated approaches such as ours could be made more accurate. However, proposing such a set of restrictions is a challenge on its own, i.e., how to create it so that designer as well as end user experience are minimally impacted, yet that it enables automated approaches based on visual similarity to protect the user in a more accurate manner. Although countermeasures of this thesis were focused on Android, such approaches could potentially be applied to other platforms as well.

7.2 Final Remarks

In this thesis, we described why the security of user interfaces is a fundamental component for the overall security of the system. Guided by the observation that users are bad at protecting themselves (e.g., in noticing security indicators), we have proposed automated countermeasures based on visual similarity as well as modeling user perception. We have shown that the mix offers unique benefits (e.g., obfuscation resilience), but our proposed techniques are not limited to detecting spoofing attacks, and could potentially be employed to detect other classes of UI attacks as well. Before our countermeasures, the attacker was incentivized to create pixel-perfect spoofing copies, and such attacks are often impossible to detect for

users. We have therefore forced the attacker to visually deviate from the perfect copies, or otherwise risk detection.

Due to the many different possible attack vectors, fully preventing user interface attacks is a challenging task. However, we can significantly raise the bar for the attacker, and this thesis is a step in that direction.

Appendices

A User Study Questions

In Chapter 5, we presented the user study we performed on how users perceive visual modifications. For the interested reader, below we list all the questions we used in our user study.

Q1: "*What is your gender?*"

- "Male", "Female"

Q2: "*How old are you?*"

- 18-29, 30-39, 40-49, 50-59, Above 60

Q3: "*What is your current education level?*"

- "Primary school", "High School", "Bachelor"

Q4: "*Do you actively use an Android device?*"

- "Yes", "No"

Q5: "*Do you use the Android Facebook application?*"

- "Yes", "No, I don't use Facebook on Android"

Q6: "*When was the last time you had to enter your password into the Android Facebook login screen?*"

- "Less than one week ago"
- "Less than one month ago"
- "More than one month ago"
- "I don't use Facebook on Android"

Q7: "*What is the Facebook application good for?*"

- "Driving a car"
- "Brushing your teeth"
- "Petting a cat"
- "Keeping in touch with friends and family"

Q8: "*Is this screen (smart phone screenshot) the Facebook login screen as you remember it?*"

- "Yes", "No"

Q9: "*How similar is this screen to the Facebook login screen you remember?*"

- with reply alternatives from 1: "Completely different" to 5: "Exactly the same"

Q10: "*If you would see this screen, how comfortable would you feel logging in?*"

- with reply alternatives from 1: "Very uncomfortable" to 5: "Very comfortable"

Q11: "*If you would see this screen, would you login with your real Facebook password?*"

- "Yes", "No"

Q12: "*In one short sentence, describe your reason for the previous answer*"

- with text input.

Additional questions for the Twitter application that has a distributed login screen.

A1: "*Is this screen (smart phone screenshot) the Twitter initial screen as you remember it?*"

- "Yes", "No"

A2: "*How similar is this screen to the Twitter initial screen you remember?*"

- with reply alternatives from 1: "Completely different" to 5: "Exactly the same"

A3: "*If you would see this screen, how comfortable would you feel clicking 'Log in'?*"

- with reply alternatives from 1: "Very uncomfortable" to 5: "Very comfortable."

A4: "*If you would see this screen, would you click 'Log in'?*"

- "Yes", "No"

Bibliography

[1] Android malware genome project. http://www.malgenomeproject.org.

[2] Google safe browsing. http://googleonlinesecurity.blogspot.com/2012/06/safe-browsing-protecting-web-users-for.html.

[3] Spoofguard. http://crypto.stanford.edu/SpoofGuard/.

[4] View. Android Developers, http://goo.gl/LKju3R.

[5] GNOME Desktop Environment. https://www.gnome.org/, Accessed April 2017.

[6] KDE Desktop Environment. https://www.kde.org/, Accessed April 2017.

[7] Tempest shielding. https://web.archive.org/web/20141229080627/netcents.af.mil/shared/media/document/AFD-140107-011.pdf, Accessed April 2017.

[8] Google glass. https://www.google.com/glass/start/, Accessed May 2017.

[9] Hardware modification. https://arstechnica.com/tech-policy/2014/05/photos-of-an-nsa-upgrade, Accessed May 2017.

[10] Trusted computer system evaluation criteria no. dod 5200.28/std. http://csrc.nist.gov/publications/history/dod85.pdf, Accessed May 2017.

[11] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009.

[12] S. Afroz and R. Greenstadt. Phishzoo: Detecting phishing websites by looking at them. In *Fifth IEEE International Conference on Semantic Computing (ICSC)*, 2011.

[13] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *International Conference on Automated Software Engineering (ASE)*, 2012.

[14] Amazon. Amazon Echo. `https://www.amazon.com/Amazon-Echo-And-Alexa-Devices/b/?node=9818047011`, Accessed April 2017.

[15] Android Developers. WindowManager. `http://developer.android.com/reference/android/view/WindowManager.html`, accessed aug. 2016.

[16] Android Developers. Manifest.permission. `http://goo.gl/3y0gpw`, accessed feb. 2016.

[17] Android Developers. Permission elements. `developer.android.com/guide/topics/manifest/permission-element.html`, accessed feb. 2016.

[18] Android Developers. Permissions. `https://developer.android.com/preview/features/runtime-permissions.html`, accessed feb. 2016.

[19] S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, Z. Sun, A. J. Blumberg, and M. Walfish. Defending against malicious peripherals with cinch. In *USENIX Security Symposium*, 2016.

[20] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[21] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context,

flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Notices*, volume 49, pages 259–269. ACM, 2014.

[22] D. Asonov and R. Agrawal. Keyboard acoustic emanations. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 3–11. IEEE, 2004.

[23] A. J. Aviv, B. Sapp, M. Blaze, and J. M. Smith. Practicality of accelerometer side channels on smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 41–50. ACM, 2012.

[24] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2013.

[25] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder. Acoustic side-channel attacks on printers. In *USENIX Security symposium*, pages 307–322, 2010.

[26] F. A. Barbhuiya, T. Saikia, and S. Nandi. An anomaly based approach for hid attack detection using keystroke dynamics. In *Cyberspace Safety and Security*, pages 139–152. Springer, 2012.

[27] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 73–84. ACM, 2010.

[28] A. M. Bates, R. Leonard, H. Pruse, D. Lowd, and K. R. Butler. Leveraging usb to establish host identity using commodity devices. In *NDSS*, 2014.

[29] BGR. Sales of Samsung's Galaxy Note lineup reportedly top 40M. `http://goo.gl/ItC6gJ`, accessed aug. 2016.

[30] BGR. Samsung: Galaxy S5 sales stronger than Galaxy S4. `http://goo.gl/EkyXjQ`, accessed aug. 2016.

[31] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the app is that? deception and countermeasures in the android user interface. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 931–948. IEEE, 2015.

[32] Biotronik. Cardiac rhythm management. `http://goo.gl/jvCuzC`.

[33] C. Bo, L. Zhang, X.-Y. Li, Q. Huang, and Y. Wang. Silentsense: silent user identification via touch and movement behavioral biometrics. In *Proceedings of the 19th annual international conference on Mobile computing & networking*, pages 187–190. ACM, 2013.

[34] G. Bradski. *Dr. Dobb's Journal of Software Tools*, 2000.

[35] J. Buchner. `https://pypi.python.org/pypi/ImageHash`.

[36] L. Cai and H. Chen. Touchlogger: Inferring keystrokes on touch screen from smartphone motion. *HotSec*, 11:9–9, 2011.

[37] L. Cai and H. Chen. On the practicality of motion based keystroke inference attack. In *International Conference on Trust and Trustworthy Computing*, pages 273–290. Springer, 2012.

[38] Z. Cai, C. Shen, and X. Guan. Mitigating behavioral variability for mouse dynamics: A dimensionality-reduction-based approach. *Transactions Human-Machine Systems*, 44(2), 2014.

[39] J. Chen, H. Chen, E. Bauman, Z. Lin, B. Zang, and H. Guan. You shouldn't collect my secrets: Thwarting sensitive keystroke leakage in mobile ime apps. In *USENIX Security*, pages 657–690, 2015.

[40] K. Chen. Reversing and exploiting an apple firmware update. In *Black Hat USA*, 2009.

[41] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *USENIX Security*, volume 15, 2015.

[42] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *USENIX Security*, volume 14, pages 1037–1052, 2014.

[43] T.-C. Chen, S. Dick, and J. Miller. Detecting visually similar web pages: Application to phishing detection. *ACM Transactions on Internet Technology (TOIT)*, 10(2):5, 2010.

[44] T. Chowdhury. *Single Microphone Tap Localization*. PhD thesis, University of Toronto, 2013.

[45] D. D. Clark and M. S. Blumenthal. The end-to-end argument and application design: the role of trust. *Fed. Comm. LJ*, 63:357, 2010.

[46] S. S. Clark, B. Ransford, A. Rahmati, S. Guineau, J. Sorber, W. Xu, and K. Fu. Wattsupdoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices. In *USENIX Workshop on Health Information Technologies (HealthTech)*, 2013.

[47] CoCoSys Endpoint Protector. `https://www.endpointprotector.com/products/endpoint-protector`, Accessed April 2017.

[48] Comodo Advanced Endpoint Protection. `https://www.comodo.com/endpoint-protection/endpoint-security.php`, Accessed April 2017.

[49] comScore. The 2015 u.s. mobile app report, 2015.

[50] A. Crenshaw. Plug and prey: Malicious usb devices, 2011.

[51] D. Damopoulos, G. Kambourakis, and S. Gritzalis. From keyloggers to touchloggers: Take the rough with the smooth. *Computers & security*, 32:102–114, 2013.

[52] G. Data. USB Keyboard Guard. `https://www.gdatasoftware.com/en-usb-keyboard-guard`, Accessed April 2017.

[53] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004.

[54] L. Davi, A.-R. Sadeghi, and M. Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51. ACM, 2011.

[55] R. Dhamija and J. D. Tygar. The battle against phishing: Dynamic security skins. In *Symposium on Usable Privacy and Security*, SOUPS'05, pages 77–88, 2005.

[56] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 581–590. ACM, 2006.

[57] W. Diao, X. Liu, Z. Li, and K. Zhang. No pardon for the interruption: New inference attacks on android through interrupt timing analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 414–432. IEEE, 2016.

[58] S. Egelman, L. F. Cranor, and J. Hong. You've been warned: an empirical study of the effectiveness of web browser phishing warnings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1065–1074. ACM, 2008.

[59] W. Enck, P. McDaniel, and T. Jaeger. Pinup: Pinning user files to known applications. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 55–64. IEEE, 2008.

[60] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88. USENIX Association, 2006.

[61] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.

[62] A. P. Felt, D. Song, D. Wagner, and S. Hanna. Android Permissions Demystified. In *Proceedings of the 18th ACM conference on Computer and Communications Security*, CCS '12, 2012.

[63] A. P. Felt and D. Wagner. *Phishing on mobile devices*. na, 2011.

[64] A. P. Felt and D. Wagner. Phishing on mobile devices. In *Web 2.0 Security and Privacy Workshop (W2SP)*, 2011.

[65] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576–587. ACM, 2014.

[66] E. Fernandes, Q. A. Chen, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash. Tivos: Trusted visual i/o paths for android. *University of Michigan CSE Technical Report CSE-TR-586-14*, 2014.

[67] E. Fernandes, Q. A. Chen, J. Paupore, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash. Android ui deception revisited: Attacks and defenses. In *Proceedings of the 20th International Conference on Financial Cryptography and Data Security*, 2016.

[68] T. Fiebig, J. Krissler, and R. Hänsch. Security impact of high resolution smartphone cameras. In *Proceedings of the 8th USENIX conference on Offensive Technologies*, WOOT '14, 2014.

[69] Forbes. Alleged 'Nazi' Android FBI Ransomware Mastermind Arrested In Russia, April 2015. `http://goo.gl/0dIrFu`.

[70] Forbes. Samsung's Galaxy Note 3 Alone Approaches 50% Of All Of Apple's iPhone Sales. `http://goo.gl/xY8t3Y`, accessed aug. 2016.

[71] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *USENIX Security Symposium*, volume 112, 2001.

[72] A. Y. Fu, L. Wenyin, and X. Deng. Detecting phishing web pages with visual similarity assessment based on earth mover's distance (emd). *IEEE transactions on dependable and secure computing*, 3(4), 2006.

[73] D. Gentner and J. Nielsen. The anti-mac interface. *Communications of the ACM*, 39(8):70–82, 1996.

[74] GFI EndPointSecurity. `https://www.gfi.com/products-and-solutions/network-security-solutions/gfi-endpointsecurity`, Accessed April 2017.

[75] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung. Vision: automated security validation of mobile apps at app markets. In *Proceedings of the second international workshop on Mobile cloud computing and services*, pages 21–26. ACM, 2011.

[76] H. Gonzalez, N. Stakhanova, and A. A. Ghorbani. Measuring code reuse in android apps. In *Privacy, Security and Trust (PST), 2016 14th Annual Conference on*, pages 187–195. IEEE, 2016.

[77] T. Goodspeed. Facedancer21, 2017. http://goodfet.sourceforge.net/hardware/facedancer21/.

[78] Google. Google Home. https://madeby.google.com/home/, Accessed April 2017.

[79] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 2012.

[80] M. Graham, T. Kubose, D. Jordan, J. Zhang, T. R. Johnson, and V. L. Patel. Heuristic evaluation of infusion pumps: implications for patient safety in intensive care units. *Journal of Medical Informatics*, 2004.

[81] K. Griffin, S. Schneider, X. Hu, and T.-C. Chiueh. Automatic generation of string signatures for malware detection. In *Recent advances in intrusion detection*, pages 101–120. Springer, 2009.

[82] F. Griscioli, M. Pizzonia, and M. Sacchetti. Usbcheckin: Preventing badusb attacks by forcing human-device interaction. In *Privacy, Security and Trust (PST), 2016 14th Annual Conference on*, pages 493–496. IEEE, 2016.

[83] A.-P. W. Group et al. Phishing activity trends report. 2006, 2009.

[84] Hak5. Usb rubber ducky, 2017. http://usbrubberducky.com/.

[85] J. Han, E. Owusu, L. T. Nguyen, A. Perrig, and J. Zhang. ACComplice: Location inference using accelerometers on smartphones. In *Proc. of the Fourth IEEE International Conference on Communication Systems and Networks*, COMSNETS '12, 2012.

[86] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 62–81. Springer, 2012.

[87] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[88] J. Hong. The state of phishing attacks. *Communications of the ACM*, 55(1), 2012.

[89] L. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson. Clickjacking revisited: A perceptual view of ui security. In *Proceedings of the USENIX Workshop on Offensive Technologies*, WOOT '14, 2014.

[90] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson. Clickjacking: Attacks and defenses. In *USENIX Security Symposium*, pages 413–428, 2012.

[91] Intel. Identity protection technology with pki - technology overview, 2013. `https://goo.gl/TtgzXW`.

[92] International Business Times. Samsung Galaxy S4 Hits 40 Million Sales Mark: CEO JK Shin Insists Device Not In Trouble Amid Slowing Monthly Sales Figures. `http://goo.gl/hU9Vdn`, accessed aug. 2016.

[93] International Secure Systems Lab. Antiphish, last access 2015. `http://www.iseclab.org/projects/antiphish/`.

[94] J. R. Jacobs. *Measuring the effectiveness of the USB flash drive as a vector for social engineering attacks on commercial and residential computer systems*. PhD thesis, Embry-Riddle Aeronautical University, 2011.

[95] M. Jakobsson, A. Tsow, A. Shah, E. Blevis, and Y.-K. Lim. What instills trust? a qualitative study of phishing. In *International Conference on Financial Cryptography and Data Security*, pages 356–361. Springer, 2007.

[96] M. Kang. *USBWall: A Novel Security Mechanism to Protect Against Maliciously Reprogrammed USB Devices*. PhD thesis, University of Kansas, 2015.

[97] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side channel cryptanalysis of product ciphers. *Computer Security—ESORICS 98*, pages 97–110, 1998.

[98] KeyGhost. The Hardware Keylogger. `http://www.keyghost.com/`, Accessed April 2017.

[99] D. Kierznowski. BadUSB 2.0: USB man in the middle attacks. https://www.royalholloway.ac.uk/isg/documents/pdf/technicalreports/2016/rhul-isg-2016-7-david-kierznowski.pdf, Accessed April 2017.

[100] D. Kopeček. USBGuard. https://dkopecek.github.io/usbguard/, Accessed April 2017.

[101] M. Kropp and P. Morales. Automated gui testing on the android platform. In *International Conference on Testing Software and Systems (ICTSS)*, 2010.

[102] M. G. Kuhn. Electromagnetic eavesdropping risks of flat-panel displays. In *International Workshop on Privacy Enhancing Technologies*, pages 88–107. Springer, 2004.

[103] M. Lange and S. Liebergeld. Crossover: secure and usable user interface for mobile devices with multiple isolated os personalities. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 249–257. ACM, 2013.

[104] L. Letaw, J. Pletcher, and K. Butler. Host identification via usb fingerprinting. In *Systematic Approaches to Digital Forensic Engineering (SADFE), 2011 IEEE Sixth International Workshop on*, pages 1–9. IEEE, 2011.

[105] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, page 8. ACM, 2014.

[106] Y. Li, J. M. McCune, and A. Perrig. Sbap: software-based attestation for peripherals. In *International Conference on Trust and Trustworthy Computing*, pages 16–29. Springer, 2010.

[107] C.-C. Lin, H. Li, X. Zhou, and X. Wang. Screenmilker: How to milk your android screen for secrets. In *Network and Distributed System Security (NDSS) Symposium*, 2014.

[108] Y.-D. Lin, Y.-C. Lai, C.-H. Chen, and H.-C. Tsai. Identifying android malicious repackaged applications by thread-grained system call sequences. *computers & security*, 39:340–350, 2013.

[109] R. Lissermann, J. Huber, M. Schmitz, J. Steimle, and M. Mühlhäuser. Permulin: mixed-focus collaboration on multi-view tabletops. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 3191–3200. ACM, 2014.

[110] J. Liu, Y. Wang, G. Kar, Y. Chen, J. Yang, and M. Gruteser. Snooping keystrokes with mm-level audio ranging on a single phone. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 142–154. ACM, 2015.

[111] W. Liu, X. Deng, G. Huang, and A. Fu. An antiphishing strategy based on visual similarity assessment. *IEEE Internet Computing*, 10(2), March 2006.

[112] X. Liu, Z. Zhou, W. Diao, Z. Li, and K. Zhang. When good becomes evil: Keystroke inference with smartwatch. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1273–1285. ACM, 2015.

[113] Liz Nardozza. USB-IF Statement regarding USB security . `http://www.usb.org/press/USB-IF_Statement_on_USB_Security_FINAL.pdf`, Accessed April 2017.

[114] E. L. Loe, H.-C. Hsiao, T. H.-J. Kim, S.-C. Lee, and S.-M. Cheng. Sandusb: An installation-free sandbox for usb peripherals. In *Internet of Things (WF-IoT), 2016 IEEE 3rd World Forum on*, pages 621–626. IEEE, 2016.

[115] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2), 2004.

[116] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du. Touchjacking attacks on web in android, ios, and windows phone. In *International Symposium on Foundations and Practice of Security*, pages 227–243. Springer, 2012.

[117] M. Campbell. Why handing Android app permission control back to users is a mistake. Tech Republic, `http://goo.gl/SYI927`, May 2015. Accessed aug. 2016.

[118] MacRumors. Masque attack vulnerability allows malicious third-party iOS apps to masquerade as legitimate apps. `http://www.macrumors.com/2014/11/10/masque-attack-ios-vulnerability/`.

[119] L. Malisa, K. Kostiainen, T. Knell, D. Sommer, and S. Capkun. Hacking in the blind:(almost) invisible runtime ui attacks on safety-critical terminals. *arXiv preprint arXiv:1604.04723*, 2016.

[120] L. Malisa, K. Kostiainen, M. Och, and S. Capkun. Mobile application impersonation detection using dynamic user interface extraction. In *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*, pages 217–237, 2016.

[121] C. Marforio, R. J. Masti, C. Soriente, K. Kostiainen, and S. Capkun. Personalized security indicators to detect application phishing attacks in mobile platforms. *arXiv preprint arXiv:1502.06824*, 2015.

[122] P. Marquardt, A. Verma, H. Carter, and P. Traynor. (sp)iphone: Decoding vibrations from nearby keyboards using mobile phone accelerometers. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 551–562. ACM, 2011.

[123] J. Maskiewicz, B. Ellis, J. Mouradian, and H. Shacham. Mouse trap: Exploiting firmware updates in usb peripherals. In *WOOT*, 2014.

[124] R. J. Masti, D. Rai, A. Ranganathan, C. Müller, L. Thiele, and S. Capkun. Thermal covert channels on multi-core platforms. In *USENIX Security*, pages 865–880, 2015.

[125] M.-E. Maurer and D. Herzner. Using visual website similarity for phishing detection and reporting. In *Extended Abstracts on Human Factors in Computing Systems (CHI)*, 2012.

[126] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 315–328. ACM, 2008.

[127] J. M. McCune, A. Perrig, and M. K. Reiter. Bump in the ether: A framework for securing sensitive user input. In *Proceedings of the annual conference on USENIX'06 Annual Technical Conference*, pages 17–17. USENIX Association, 2006.

[128] J. M. McCune, A. Perrig, and M. K. Reiter. Safe passage for passwords and other sensitive data. In *NDSS*, 2009.

[129] E. Medvet, E. Kirda, and C. Kruegel. Visual-similarity-based phishing detection. In *International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2008.

[130] A. Memon, I. Banerjee, and A. Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *null*, page 260. IEEE, 2003.

[131] W. Metzger. *Laws of Seeing*. The MIT Press, 2009.

[132] Y. Michalevsky, D. Boneh, and G. Nakibly. Gyrophone: Recognizing speech from gyroscope signals. In *USENIX Security*, pages 1053–1067, 2014.

[133] Microsoft. Lockdown features (windows embedded industry 8.1), 2014. `https://goo.gl/JcXC9X`.

[134] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury. Tapprints: your finger taps have fingerprints. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 323–336. ACM, 2012.

[135] S. Mondal and P. Bours. A computational approach to the continuous authentication biometric system. *Information Sciences*, 304(C), 2015.

[136] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna. Baredroid: Large-scale analysis of android apps on real devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 71–80. ACM, 2015.

[137] S. Narain, A. Sanatinia, and G. Noubir. Single-stroke language-agnostic keylogging using stereo-microphones and domain specific machine learning. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 201–212. ACM, 2014.

[138] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon. Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, 21(1), 2014.

[139] M. Niemietz and J. Schwenk. Ui redressing attacks on android devices. *Black Hat Abu Dhabi*, 2012.

[140] K. Nohl and J. Lell. Badusb – on accessories that turn evil. In *Black Hat USA*, 2014.

[141] J. Oberheide and C. Miller. Dissecting the android bouncer. *SummerCon2012, New York*, 2012.

[142] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang. Accessory: password inference using accelerometers on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, page 9. ACM, 2012.

[143] PAX Team. Pax address space layout randomization. `http://pax.grsecurity.net/docs/aslr.txt`, Accessed April 2017.

[144] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[145] T. Perković, M. Čagalj, and N. Rakić. Sssl: shoulder surfing safe login. 2010.

[146] D. Ping, X. Sun, and B. Mao. Textlogger: inferring longer inputs on touch screen using motion sensors. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, page 24. ACM, 2015.

[147] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 209–220. ACM, 2013.

[148] Red Hat. SELinux Project. `https://selinuxproject.org/page/Main_Page`, Accessed April 2017.

[149] C. Ren, P. Liu, and S. Zhu. Windowguard: Systematic protection of gui security in android. 2017.

[150] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards discovering and understanding task hijacking in android. In *USENIX Security*, pages 945–959, 2015.

[151] R. A. Rensink. Change detection. *Annual review of psychology*, 53(1), 2002.

[152] F. Roesner and T. Kohno. Securing embedded user interfaces: Android and beyond. In *USENIX Security*, pages 97–112, 2013.

[153] F. Roesner and T. Kohno. Improving Accuracy, Applicability and Usability of Keystroke Biometrics on Mobile Touchscreen Devices. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, 2015.

[154] A. P. Rosiello, E. Kirda, C. Kruegel, and F. Ferrandi. A layout-similarity-based approach for detecting phishing pages. In *Conference on Security and Privacy in Communications Networks (SecureComm)*, 2007.

[155] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. Orb: An efficient alternative to sift or surf. In *International Conference on Computer Vision (ICCV)*, 2011.

[156] G. Rydstedt, B. Gourdin, E. Bursztein, and D. Boneh. Framing attacks on smart phones and dumb routers: Tap-jacking and geo-localization attacks. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, WOOT'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[157] H. Sahbi, L. Ballan, G. Serra, and A. Del Bimbo. Context-dependent logo matching and recognition. *Image Processing, IEEE Transactions on*, 22(3), March 2013.

[158] SAMSUNG. Samsung GALAXY S4. `http://goo.gl/R32WhA`, Accessed August 2016.

[159] A. Sarkisyan, R. Debbiny, and A. Nahapetian. Wristsnoop: Smartphone pins prediction using smartwatch motion sensors. In *Information Forensics and Security (WIFS), 2015 IEEE International Workshop on*, pages 1–6. IEEE, 2015.

[160] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The emperor's new security indicators. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 51–65. IEEE, 2007.

[161] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: a stealthy and context-aware sound trojan for smartphones. In *NDSS*, volume 11, pages 17–33, 2011.

[162] M. Selhorst, C. Stüble, F. Feldmann, and U. Gnaida. Towards a trusted mobile desktop. In *International conference on Trust and trustworthy computing*, pages 78–94. Springer, 2010.

[163] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. "andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.

[164] C. Shen, Z. Cai, X. Guan, Y. Du, and R. Maxion. User authentication through mouse dynamics. *Information Forensics and Security, IEEE Transactions on*, 8(1), 2013.

[165] C. Shen, Z. Cai, X. Guan, H. Sha, and J. Du. Feature analysis of mouse dynamics in identity authentication and monitoring. In *IEEE International Conference on Communications (ICC)*, 2009.

[166] P. Shrestha, M. Mohamed, and N. Saxena. Slogger: Smashing motion-based touchstroke logging with transparent system noise. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 67–77. ACM, 2016.

[167] L. Simon and R. Anderson. Pin skimmer: Inferring pins through the camera and microphone. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*, pages 67–78. ACM, 2013.

[168] D. J. Simons and R. A. Rensink. Change blindness: past, present, and future. *TRENDS in Cognitive Sciences*, 9(1), 2005.

[169] Y. Song, M. Kukreti, R. Rawat, and U. Hengartner. Two novel defenses against motion-based keystroke inference attacks. *arXiv preprint arXiv:1410.7746*, 2014.

[170] SONY Developer World. Floating Touch. `developer.sonymobile.com/knowledge-base/technologies/floating-touch/`, Accessed August 2016.

[171] Spider Labs. Focus stealing vulnerability in android. http://blog.spiderlabs.com/2011/08/twsl2011-008-focus-stealing-vulnerability-in-android.html.

[172] R. Spreitzer. Pin skimming: Exploiting the ambient-light sensor in mobile devices. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pages 51–62. ACM, 2014.

[173] L. Stefanko. Android Banking Trojan, March 2016. http://www.welivesecurity.com/2016/03/09/android-trojan-targets-online-banking-users/.

[174] G. P. Store. Lockdown pro. https://play.google.com/store/apps/details?id=appplus.mobi.lockdownpro.

[175] M. Sun, M. Li, and J. Lui. Droideagle: seamless detection of visually similar android apps. In *Conference on Security and Privacy in Wireless and Mobile Networks (Wisec)*, 2015.

[176] Symantec. Will Your Next TV Manual Ask You to Run a Scan Instead of Adjusting the Antenna?, April 2015. https://goo.gl/vMZlGX.

[177] Z. Tang, Y. Dai, and X. Zhang. Perceptual hashing for color images using invariant moments. *Appl. Math*, 6(2S):643S–650S, 2012.

[178] R. Templeman, Z. Rahman, D. Crandall, and A. Kapadia. Placeraider: Virtual theft in physical spaces with smartphones. *arXiv preprint arXiv:1209.5982*, 2012.

[179] D. J. Tian, A. Bates, and K. Butler. Defending against malicious usb firmware with goodusb. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 261–270. ACM, 2015.

[180] D. J. Tian, N. Scaife, A. Bates, K. Butler, and P. Traynor. Making usb great again with usbfilter. In *Proceedings of the USENIX Security Symposium*, 2016.

[181] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, 2002.

[182] M. Tischer, Z. Durumeric, S. Foster, S. Duan, A. Mori, E. Bursztein, and M. Bailey. Users really do plug in usb drives they find. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 306–319. IEEE, 2016.

[183] T. Tong and D. Evans. Guardroid: A trusted path for password entry. *Proceedings of Mobile Security Technologies (MoST)*, 2013.

[184] T. Vidas, J. Tan, J. Nahata, C. L. Tan, N. Christin, and P. Tague. A5: Automated analysis of adversarial android applications. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. ACM, 2014.

[185] E. von Zezschwitz, A. Koslow, A. De Luca, and H. Hussmann. Making graphic-based authentication secure against smudge attacks. In *International Conference on Intelligent User Interfaces (IUI)*, 2013.

[186] H. Wang, T. T.-T. Lai, and R. Roy Choudhury. Mole: Motion leaks through smartwatch sensors. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 155–166. ACM, 2015.

[187] Z. Wang and A. Stavrou. Exploiting smart-phone usb connectivity for fun and profit. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 357–366. ACM, 2010.

[188] Z. Wang and A. Stavrou. Attestation & authentication for usb communications. In *Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on*, pages 43–44. IEEE, 2012.

[189] S. Weiser and M. Werner. Sgxio: Generic trusted i/o path for intel sgx. *arXiv preprint arXiv:1701.01061*, 2017.

[190] M. Wertheimer and K. Riezler. Gestalt theory. *Social Research*, pages 78–99, 1944.

[191] L. Wu, X. Du, and J. Wu. Effective defense schemes for phishing attacks on mobile computing platforms. *IEEE Transactions on Vehicular Technology*, 2015.

[192] M. Wu, R. C. Miller, and S. L. Garfinkel. Do security toolbars actually prevent phishing attacks? In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 601–610. ACM, 2006.

[193] X. Wu and G. Zhai. Temporal psychovisual modulation: A new paradigm of information display [exploratory dsp]. *IEEE Signal Processing Magazine*, 30(1):136–141, 2013.

[194] G. Xiang, J. Hong, C. P. Rose, and L. Cranor. Cantina+: A feature-rich machine learning framework for detecting phishing web sites. *ACM Transactions on Information and System Security (TISSEC)*, 14(2):21, 2011.

[195] H. Xu, Y. Zhou, and M. R. Lyu. Towards Continuous and Passive Authentication via Touch Biometrics: An Experimental Study on Smartphones. In *Proceedings of the Symposium On Usable Privacy and Security*, SOUPS '14, 2014.

[196] Z. Xu, K. Bai, and S. Zhu. Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 113–124. ACM, 2012.

[197] Z. Xu and S. Zhu. Abusing notification services on smartphones for phishing and spamming. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2012.

[198] L. K. Yan and H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 569–584, 2012.

[199] W. Yang, M. Prasad, and T. Xie. A grey-box approach for automated gui-model generation of mobile applications. In *Fundamental Approaches to Software Engineering (FASE)*. 2013.

[200] Q. Yue, Z. Ling, X. Fu, B. Liu, K. Ren, and W. Zhao. Blind recognition of touched keys on mobile devices. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1403–1414. ACM, 2014.

[201] H. Zadgaonkar. *Robotium Automated Testing for Android*. Packt Publishing, 2013.

[202] C. Zauner. *Implementation and benchmarking of perceptual image hash functions*. na, 2010.

[203] J. Zhai and J. Su. The service you can't refuse: A secluded hijackrat. https://www.fireeye.com/blog/threat-research/2014/07/the-service-you-cant-refuse-a-secluded-hijackrat.html.

[204] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *ACM conference on Security and privacy in wireless and mobile networks (Wisec)*, 2014.

[205] H. Zhang, G. Liu, T. Chow, and W. Liu. Textual and visual content-based anti-phishing: A bayesian approach. *IEEE Transactions on Neural Networks*, 22(10), Oct 2011.

[206] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang. Leave me alone: App-level protection against runtime information gathering on android. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 915–930. IEEE, 2015.

[207] Q. Zhang and D. S. Reeves. Metaaware: Identifying metamorphic malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 411–420. IEEE, 2007.

[208] Y. Zhang, J. I. Hong, and L. F. Cranor. Cantina: A content-based approach to detecting phishing web sites. In *International Conference on World Wide Web (WWW)*, 2007.

[209] Y. Zhang, P. Xia, J. Luo, Z. Ling, B. Liu, and X. Fu. Fingerprint attack against touch-enabled devices. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 57–68. ACM, 2012.

[210] N. Zheng, K. Bai, H. Huang, and H. Wang. You are how you touch: User verification on smartphones via tapping behaviors. In *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*, pages 221–232. IEEE, 2014.

[211] N. Zheng, A. Paloski, and H. Wang. An efficient user verification system via mouse movements. In *Computer and Communications Security (CCS)*, 2011.

[212] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the*

third ACM conference on Data and application security and privacy, pages 185–196. ACM, 2013.

[213] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*. ACM, 2012.

[214] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

[215] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.

[216] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building verifiable trusted path on commodity x86 computers. In *IEEE Symposium on Security and Privacy*, S&P'12, pages 616–630, 2012.

[217] L. Zhuang, F. Zhou, and J. D. Tygar. Keyboard acoustic emanations revisited. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):3, 2009.

Resume

Luka Mališa
Born in Šibenik, Croatia on December 31, 1984
Citizen of Croatia

Education

2011 – 2017	**Doctor of Sciences** ETH Zurich
2009 – 2011	**Master of Science in Computer Science** ETH Zurich
2003 – 2008	**Bachelor in Computer Science** Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture (FESB), Split, Croatia

Work Experience

2011 – 2017	**Research Assistant** Institute for Information Security, ETH Zurich