SECURING MOBILE PHONES IN THE EVOLVING MOBILE ECOSYSTEM

A Dissertation in
Computer Science and Engineering
by
Machigar Ongtang

© 2010 Machigar Ongtang

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2010
The dissertation of Machigar Ongtang was reviewed and approved* by the following:

Patrick D. McDaniel
Associate Professor of Computer Science and Engineering
Dissertation Advisor, Chair of Committee

Thomas F. La Porta
Distinguished Professor of Computer Science and Engineering

Trent Jaeger
Associate Professor of Computer Science and Engineering

Ling Rothrock
Associate Professor of Industrial and Manufacturing Engineering

Raj Acharya
Professor of Computer Science and Engineering
Department Head

*Signatures are on file in the Graduate School.
Abstract

The revolution of mobile phone industry has been altering our life and business practices over the past few decades. Driven by user demands and technological advancement, we now experience rich mobile phone applications and sophisticated cellular services ranging from mobile payment, stock trading, to social networking, vehicle tracking to in-car control. As more players joining the community, this mobile phone environment has transformed into a complex network of interacting companies, known as mobile ecosystem. Unfortunately, this opening and converging mobile ecosystem has brought in more opportunities for more attacks on mobile phones, a key element of the system.

This dissertation aims to achieve mobile phone security. We reveal two main challenges that we need to overcome, namely developing a clear definition of secure phones, and building security infrastructure that imposes such definition on the phones. We also identify three key elements that contribute to the fidelity of mobile phones, namely, mobile phone platforms, mobile phone applications, and mobile content. Throughout this dissertation, we explore the application of security requirements engineering discipline in developing security policies that frame security phones from the perspectives of the above three elements. Accordingly, we develop appropriate security infrastructure on the Android platform to enforce such security policies. Through security analysis and performance evaluation, we demonstrate the effectiveness and the efficiency of our approach, and suggest possible further improvement.
Table of Contents

List of Figures viii

List of Tables x

Acknowledgments xi

Chapter 1
Introduction 1
1.1 Thesis Statement .............................................. 5
1.2 Contributions .................................................. 7
1.3 Dissertation Outline ........................................... 8

Chapter 2
Preliminaries and Related Work 10
2.1 Mobile Ecosystem and its Evolution ........................ 11
  2.1.1 Evolution of the Mobile Ecosystem ....................... 11
  2.1.2 An Outlook for Mobile Phone Security ................... 13
2.2 Android Operating System ................................... 15
  2.2.1 Android Architecture ...................................... 16
  2.2.2 Android Application Model ............................... 16
  2.2.3 Android Security Model ................................... 18
  2.2.4 Limitations of Android .................................... 20
2.3 Basic Mobile Phone Operations .............................. 21
  2.3.1 Short Messaging Service (SMS) .......................... 22
  2.3.2 Multimedia Messaging Service (MMS) .................... 22
  2.3.3 Internet Connection ....................................... 24
2.4 Attacks on Mobile Phones ................................... 24
  2.4.1 Cellular Network Vulnerabilities ......................... 25
  2.4.2 Mobile Phone Vulnerabilities ........................... 25
2.5 Attack Mitigations ................................................. 27
2.6 Security Requirements Engineering (SRE) ...................... 30
2.7 Policy Systems .................................................... 33
  2.7.1 Classic Policy Models ...................................... 34
  2.7.2 Decentralized Policy ...................................... 35
  2.7.3 Policy Reconciliation ...................................... 38
  2.7.4 Automated Policy Generation .............................. 38
2.8 Summary .......................................................... 39

Chapter 3
Securing Mobile Phone Platform Against Malware 40
3.1 Understanding Malware Situation ............................... 42
  3.1.1 Categories of Malware ...................................... 42
  3.1.2 Malware Mitigations and Limitations ................. 44
3.2 Certifying Application at Install Time ...................... 45
  3.2.1 Application Installation in Android .................. 45
  3.2.2 Application Certification with Kirin .................. 47
  3.2.3 Implementation of Kirin .................................. 49
3.3 Developing Platform Security Policy ......................... 49
  3.3.1 Security Requirements Engineering for Mobile Phones ........... 50
  3.3.2 Platform Security Policy Development Methodology ............. 50
  3.3.3 Sample Platform Security Policies .................. 53
  3.3.4 Kirin Security Language .................................. 58
3.4 Evaluating Platform Protection Against Malware ............ 60
  3.4.1 Empirical Results ........................................ 60
  3.4.2 Mitigating Malware ....................................... 63
  3.4.3 Discovered Vulnerabilities .............................. 63
3.5 Summary .......................................................... 65

Chapter 4
Securing Mobile Phone Applications in Open and Converging Model 66
4.1 Evolving Application Model .................................... 67
4.2 Protecting Applications’ Interactions ......................... 69
  4.2.1 Motivating Example ....................................... 69
  4.2.2 Designing Saint Framework ............................... 71
    4.2.2.1 Install-time Application Policy Support .............. 71
    4.2.2.2 Runtime Application Policy Support ................ 72
4.3 Developing Application Security Policy ...................... 73
  4.3.1 Application Security Policy Development Methodology ........ 74
  4.3.2 Sample Application Security Policy .................. 77
  4.3.3 Application Security Policy Taxonomy ................. 79
    4.3.3.1 Install-time Permission Granting Policy ............. 80
    4.3.3.2 Run-time Interaction Policy ....................... 80
Chapter 4

4.4 Auxiliary Policies .................................................. 83
  4.4.1 Administrative Policy ...................................... 83
  4.4.2 Operational Policy ......................................... 84
4.5 Implementing Saint Framework ................................. 85
  4.5.1 Saint Installer ............................................... 85
  4.5.2 Saint Mediator ............................................. 86
  4.5.3 AppPolicy Provider ....................................... 89
  4.5.4 FrameworkPolicyManager ............................... 90
  4.5.5 Condition Extensibility ................................. 90
4.6 Evaluating Applications Protection ............................ 90
  4.6.1 Policy Appropriateness .................................. 91
  4.6.2 Performance Evaluation ................................ 95
    4.6.2.1 Macrobenchmarks ................................ 96
    4.6.2.2 Microbenchmarks ................................ 97
4.7 Summary .......................................................... 97

Chapter 5

Secure Handling of Mobile Content ............................... 100
5.1 Recognizing Risks to Content and Need for DRM ............... 102
  5.1.1 Understanding Content Usage Through Applications Survey .... 104
  5.1.2 DRM Policy Requirements ............................... 106
5.2 Understanding Content Delivery ................................ 106
  5.2.1 Documents in Transit .................................... 107
  5.2.2 On-Platform Document Access ........................... 108
5.3 Threat and Trust Model ......................................... 110
5.4 Securing Mobile Content with Porscha ......................... 111
  5.4.1 Supported Policy ......................................... 111
  5.4.2 Document-in-Transit .................................... 112
  5.4.3 On-platform Document Security .......................... 115
    5.4.3.1 Policy Enforcement on Initial Recipients .......... 116
    5.4.3.2 Policy Enforcement on Documents at Rest .......... 118
    5.4.3.3 Enforcement on Indirect Receivers ................ 118
5.5 Developing Content Security Policy ............................ 119
  5.5.1 Content Policy Development Methodology ................. 119
  5.5.2 Content Security Policy Rules ........................... 122
  5.5.3 Sample Content Security Policy ......................... 122
5.6 Evaluating Porscha ............................................... 123
5.7 Discussion ....................................................... 125
  5.7.1 Protecting the Private Key .............................. 125
  5.7.2 Recipients Without Porscha ............................. 126
  5.7.3 Application and Platform Trust ......................... 126
  5.7.4 Alternative Application Enforcement Infrastructures .... 127
| 5.7.5 Digital Rights Management | 127 |

### Chapter 6

**Towards Secure Mobile Phone Environment**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Security Policy Framework</td>
<td>130</td>
</tr>
<tr>
<td>6.2 Retrofitting Security through Security Requirements Engineering</td>
<td>131</td>
</tr>
<tr>
<td>6.3 Trust Model</td>
<td>132</td>
</tr>
<tr>
<td>6.4 Future Work</td>
<td>133</td>
</tr>
<tr>
<td>6.4.1 Confining Applications with AppJailing</td>
<td>133</td>
</tr>
<tr>
<td>6.5 Cloud Computing Infrastructure for Securing Elastic Android Apps</td>
<td>136</td>
</tr>
<tr>
<td>6.6 On-Platform DRM for Application Usage Control</td>
<td>138</td>
</tr>
<tr>
<td>6.7 Summary</td>
<td>139</td>
</tr>
</tbody>
</table>

### Bibliography

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>141</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Mobile ecosystem consists of multiple interacting classes of participants 12
2.2 Android architecture consists of three layers: Linux kernel, Android middleware, and applications. .......................... 17
2.3 Android’s application manifest file contains the application’s security configuration and the component structure. ..................... 17
2.4 Android middleware mediates access between components and enforces mandatory access control (MAC) policy. As application 1 possesses label $l_1$, component A in application 1 can access component B protected by label $l_1$. However, it cannot access component C assigned with label $l_2$. 19
2.5 Simplified SMS network architecture ........................................ 22
2.6 MMS Client submits multimedia message to associated MMS Proxy Relay 24
2.7 MMS Client retrieves multimedia message after receiving the WAP Push notification .................................................. 24
3.1 Kirin based application installer performs application certification at install time. ................................................................. 48
3.2 Steps for Developing Phone Platform Security Policy. .................. 51
3.3 Problem diagram explains possible functions on user conversation .... 54
3.4 Threat tree with the global goal to eavesdrop the user’s microphone input during the conversation ................................. 55
3.5 KSL syntax in BNF. ................................................................. 58
4.1 The PersonalShopper application finds desired items at the discretion of the user and interacts with vendors and payment applications to purchase them. ............................................................. 70
4.2 Saint enforcement - Saint enhances the application installation process (a-c) with additional permission granting policies and mediates component IPC (1-5) to enforce interaction policies specified by both the caller and callee applications. .................................................... 72
4.3 Steps for Applications Security Policies Development .................. 75
4.4 Problem diagram describes PersonalShopper’s payment function which involves interacting with a payment application. ......................... 78
4.5 The attack tree for the threat which the malicious application reveals payment information to the adversary. ................................................. 78
4.6 Policy tree illustrating the example policies that are required by applications. The boxed with double stroke represent the policies already supported by the original Android platform. ............................... 79
4.7 **Saint authorization hook placement.** The added hooks for each of the four types of component interaction are numbered (1)-(8). ............... 86
4.8 Microbenchmark reveals underlying cost of Saint execution ............ 97
4.9 AppPolicy database query overhead ........................................... 98

5.1 External providers cannot control how documents are consumed inside of the mobile phones ......................................................... 102
5.2 The external providers deliver their documents to the application(s) on the phone. The shaded area represents our TCB. ...................... 107
5.3 IBE System for SMS/MMS Delivery ............................................. 113
5.4 IBE System for Email Delivery .................................................... 113
5.5 The Porscha mediator implemented inside Android intercepts the document delivery and enforces the policy on direct receivers. .......... 116
5.6 Porscha Mediator intercepts cross-application content passing ........ 118
5.7 Content Security Policy Development Methodology ...................... 120
5.8 BNF of Porscha content security policy ........................................ 123
List of Tables

3.1 Sample security rules for malware mitigation .......................... 56
3.2 Applications failing Rule 2 ..................................................... 60
3.3 Applications failing Rule 4 and 5 ........................................... 62
3.4 Unchecked Permissions (Experiment on SDK 1.1 Release 1) .......... 65

4.1 Sample security policies for secure application interactions ............ 81
4.2 Sample OpenIntents Intent action strings with service provider and con-
sumer applications ............................................................... 92
4.3 Example Scenarios for Saint’s Performance Overhead .................. 96
4.4 Performance overhead of obtaining information for policy evaluation . 98

5.1 Number of sample applications that access SMS, MMS, and email. .... 104
5.2 Examples of security policies for content protection .................... 124
5.3 Overheads in processing SMS, MMS, and Email (in seconds) ........ 125
Acknowledgments

Our life cannot be successful without contributions from individuals around us. I am very fortunate to be surrounded by incredible people. I would like to use this space to gratefully extend my thanks to those who made this dissertation possible and gave me magnificent Ph.D. years.

My arrival at Penn State would not have been possible without great support and encouragement from Dr. Warakorn Samkoset, Dr. Nucharee Premchaiswadi, and Dr. Tatree Nontasak at Dhurakij Pundit University. My ability to understand research is a testimony to Dr. Ali Hurson, my first Ph.D. advisor. Dr. Hurson taught me to think as a researcher and has guided me from the preliminary and throughout the first two years of my Ph.D..

I do owe my deepest gratitude to my advisor, Dr. Patrick McDaniel. Patrick brought me into the SIIS Lab and introduced me to the area of computer security which constitutes this dissertation. He worked very closely with me, made sure that I am on the right track, and carried me through difficulties. His high-level vision and creative thinking have motivated me to push my research to a new dimension. More importantly, Patrick showed me my strength and taught me to trust in what I do. He is both an exceptional advisor and a great mentor. He supports his students research-wise, coaches us to be professional, and also guides us in nearly every aspect of life.

I offer my regards and blessings to all of my colleagues in SIIS Lab. They have been great support and have made my Ph.D. years memorable. I am pleased to express special thanks to William Enck who inspired my research topic and helped me settle down in the lab. Research discussion with him has always been invaluable. Stephen McLaughlin deserves great recognition for supporting my research and patiently helping me with my writing. I also would like to credit Kevin Butler for his contribution to my research and for his useful comments. Moreover, my Ph.D. experience could not have been wonderful without my best friends, Divya Muthukumaran and Sandra Rueda who always listened to me and encouraged me during my hard moments.

My success is dedicated to my family. My parents and my sister have been a great support since the first time I mentioned about getting a Ph.D. It is their love and understanding that made me strong and able to stand up again in the face of failure. They are the best piece of my life and I really hope that they are proud of me.
Lastly, I would like to thank Satama Sirivunnabood, my boyfriend and best friend. In spite of his constant support, my Ph.D. years would not have begun without his question: “Wouldn’t you be sorry if you had a chance and capability to do a Ph.D. but you didn’t go for it? . . . Are you sure that you would have this chance in your next life?” It was this question that surprisingly pushed me, who never had ambition to get a Ph.D. in anything, to think about it. It has been nearly six years since that time; and now I am writing the last piece of my Ph.D. dissertation.
Chapter 1

Introduction

The mobile telecommunication industry persistently and rapidly expands. Contradicting the recent economic downturn, the number of mobile cellular subscriptions worldwide has reached an estimated 4.6 billion in 2009 [1]. It is also projected that the revenue of this lucrative business would increase from approximately $350 billion in 2002 to over $800 billion in 2011 [2]. These striking numbers reflect the continual demand for mobile telephony. The use of mobile phones has long transcended strictly voice calling. Live measurements showed that mobile data traffic has taken over voice traffic volume [3]. Global SMS traffic alone exceeded 5.5 trillion messages in 2009 [4]. As users perform ever more data-centric activities on their phones, they are increasingly purchasing smartphones with the ability to run wide varieties of applications. Over 170 million smartphones were purchased globally in 2009 [5]. Smartphones then become main conduits for various activities, from personal affairs such as diaries, journals, to mobile shopping and SMS ticketing, to financial transactions, online stock trading, community services, location-based services, and social networking. The wide adoption of smartphones continues to drive global economics, accelerating the growth across all market segments. The value of mobile payments market is expected to rise from $170 billion in 2010 to $630 billion in 2014 [6].

Unequivocally, the economic value that these mobile activities carry puts a premium on the mobile phone system. While increasingly cutting-edge functionality offered by this flourishing system attracts more players to invest in and become part of the network, so too adversaries. Evidently, these emerging applications and services open the opportunities for more attacks with potentially greater returns. We, as security researchers, must therefore investigate the use of mobile phone applications and services of today and
tomorrow to bring out security and privacy issues, and devise appropriate security solutions. Unfortunately, the enormous size and the complexity of this environment make achieving security a difficult task.

The foremost challenge in accomplishing mobile phone security is to define *secure mobile phones*. The lack of a clear notion of security in the context of mobile phones is not different from starting a voyage without a destination. Ideally, we seek for necessary and sufficient conditions for mobile phones to be secure. To define the goal for our journey, we start out with a classical definition of security. A “secure system” is one we can depend on to behave as expected [7]. In other words, the conditions under which the phones considered secure depend on the expectations of the stakeholders involved in the mobile phone system.

Mobile phone environment can be viewed as a *mobile ecosystem*, which is characterized by a large and complex network of companies interacting with each other, directly and indirectly, to provide a broad array of mobile products and services to end-customers [8]. From network operators, platform developers, application developers, content and service providers, to consumers, these are the stakeholders who influence and are impacted by the behaviors of the mobile phone system. Playing different roles in the ecosystem and having different levels of concern, the expectations these stakeholders have on mobile phones are diverse. As a result, building security to mobile phones requires the understanding of the mobile phone operating environment, the elements on the phones which are significant to the stakeholders’ activities, and the set of security requirements they would have on them.

Consider a mobile phone-centric model, the fidelity of the activities on the mobile phones involves at least three elements: mobile phone platform, mobile phone applications, and mobile content. The mobile phone platform is the fundamental of every single operation on the phones. It manages resources including hardware, network interfaces, system and application data, as well as all processes. Compromised phone platforms are detrimental to almost all of the stakeholders. They compromise applications and in some cases cost phone users financial loss arising from fraud. Sufficient number of misbehaved phones can also potentially degrade the network [9]. Mobile phone applications directly handle transactions and data. The correctness and dependability of the mobile activities as well as the confidentiality and integrity of their data rely on the integrity of the applications that operate them. More importantly, the recent trend in application collaboration encourages multiple applications to work together to create more sophisticated functionality. Even though this use and extend development model gives rise to an
explosive number of innovative applications, it propagates the effects of any exploits since a malicious or compromised application can pollute other applications. Both the phone platform and applications regularly receive various types of content such as SMS, MMS, emails, instant messages, and application-specific data, from external entities through either the cellular data network or the Internet. These content and service providers and the phone users are the stakeholders that place their trust on the phones to handle the content. Some pieces of content are sensitive. Mishandling them can expose confidential information, breach privacy, or render the applications mischievous or unusable.

Accordingly, achieving mobile phone security requires building security into these three basic building blocks, namely the phone platform, the mobile phone applications, and the mobile content. From these three perspectives, we must explore the set of conditions that must be satisfied to meet the stakeholders’ expectations. These can be materialized as the set of security requirements which the stakeholders project on these elements; analogous to the milestones on the path to our destination.

A well-known principle for stakeholders’ security requirements identification is security requirements engineering (SRE). It seeks to understand the system’s functional requirements, assets, and threats in order to develop the required constraints, which are the security requirements on the system, to protect it against the threats. This dissertation demonstrates how the SRE practice can be applied to mobile phone paradigm to acquire the stakeholders’ security requirements for the above three domains. We develop a three-prong security requirements methodology to reflect the security requirements on the mobile phone platform, the mobile phone applications, and the mobile content.

Nevertheless, security requirements engineering is up to now put input practice at organization or inter-organization level or applied in product development. We have not witnessed any adoption of SRE in securing mobile phones. We observe that even though there are several existing security requirements engineering approaches, they cannot be directly applied to the mobile phone environment due to the following four main constraints:

- There are a multitude of stakeholders. Moreover, some of them are unknowable at the time of developing the system. The set of the stakeholders are also dynamic. There are always more stakeholders joining the system after the phones are released. As a result, it is difficult (if not impossible) to acquire the complete set of security requirements and include them during that time.

- As the security requirements have not been included in the development process
since the beginning, our security requirements engineering methodology involves post-facto analysis. Most existing methodologies are designed to be integrated into the system and software development process; thus cannot be used in this environment.

- Because the stakeholders are independent, their internal functionality is usually invisible to other stakeholders. Therefore, the stakeholders security requirements are usually grounded on their conjectures about others.

- In the mobile phone environment, certain types of security requirements cannot be fulfilled by the stakeholders but only by other entities external to themselves. Due to this constraint, it is necessary to take the limitations of the capability of such entities into account.

Our proposed security requirements engineering methodology for the mobile phones addresses the above four challenges. It is based on two main technologies. First, we adapt Security Requirements Engineering Framework which will be introduced in Section 2.6. This scheme conforms to the mobile phone environment because it centers around assets rather than system processes and information flows. As the usage model of mobile phones is considerably diverse, it is simpler to manipulate the phone assets than to derive the work processes. Second, we apply threat (attack) tree, which will also be described in Section 2.6. In fact, this approach is commonly used for threat modeling rather than in security requirements engineering. However, since we work on existing phone platforms, it is a useful tool for decomposing the potential threats especially those resulted from the interactions across different functions. More specifically, we use the threat trees to determine the possible threats to the assets collected from the Problem Frame approach.

The accompanying challenge from defining security through security requirements involves the degree to which the stakeholders are allowed to specify and impose their own expectation, namely their security requirements. Traditional adoption of SRE integrates the SRE process with the system development process, extracting the requirements from the stakeholders to include adequate security mechanisms into the system design. Unfortunately, incorporating the stakeholders’ security requirements into mobile phone during its built is potentially infeasible because of the above four constraints. A solution is to provide a means for the stakeholders to impose their security requirements on the mobile phones at run time. In this scheme, the security requirements can be expressed in terms of security policies, which describe what behavior must or must not take place in order to define the secure states of the system [10]. With our three-pronged approach, we identify
the three types of security policies from the key stakeholders, namely platform security policies from the consumers and the platform developers, application security policies from the application developers, and content security policies from the content providers. However, the current mobile phones accept only limited input from the stakeholders’ in building security. We urge the need for security infrastructure that admits and enforces the stakeholders’ security policies, as an approach to incorporate their security requirements with the mobile phone security mechanisms in order to fulfill their expectations and achieve secure mobile phones. Each of the three prongs in our SRE methodology together with the security infrastructure to enforce the corresponding type of security policy is presented in Chapter 4, 5, and 6.

1.1 Thesis Statement

The growth of the mobile telecommunication industry is truly dramatic. People place their trust on the system to handle more number of increasingly critical transactions. However, the system is up to this time plagued with security problems. The growing size and degree of complexity of this system will manifest the impact of such concerns. In response, we are in need of a systematic way to approach security for mobile phones.

In accordance with the way mobile phone activities operate, there are at least three fundamental components that contribute to the fidelity of the mobile phones and that would be the center of the stakeholders’ security interests, namely, the mobile phone platform, the mobile phone applications, and the mobile content. In response, our research addresses security from the perspectives of these three basic building blocks.

The mobile phone system of today and tomorrow engages many stakeholders including, the platform providers, the network providers, the application developers, the content and service providers, and the users. The definition of secure mobile phones from the viewpoint of each of these players can be different due to the diverse set of expectations each one has on the behaviors of the phones. To achieve security, it is necessary to take their security interests into consideration. We explore the security requirements engineering research discipline to seek for an appropriate methodology for identifying the stakeholders’ security requirements for mobile phones, and express them as security policies. To realize the security requirements in the mobile phone environment which involves a large number of stakeholders, potentially unknowable at design time, we develop a security infrastructure that accepts and imposes the stakeholders’ security policies at operation time.
A chief design decision in building such infrastructure is the placement of the security decision and enforcement point. We opt to locate them on the phone platform based on the following justification. First, embedding the security policy enforcement mechanism within the platform enables tamperproofness. It cannot be modified without corrupting the operating system. Second, the enforcement must be mandatory. Including it in the platform ensures complete mediation. Third, the platform has direct contact with and control over the behaviors of the applications. It also interacts with the external entities including the networks and the users. Therefore, the platform layer contains the most semantics of the ongoing operations, which is crucial for the security decisions. Fourth, the phone platform is expected to take the brokerage role for the mobile ecosystem. It is also in a prime position to become the key hub by driving and enabling the integration of application, software, content, gaming, and media [8]. Being the rendezvous, the platform would not only have a broad picture of the system but would also become the best candidate for acquiring the security requirements from other members of the ecosystem. Lastly, as we are steering towards open platform, hardening mobile phone platform becomes more practical. Our research would serve the objective of open sourcing by reinforcing the security of the mobile phone operating system to enable better and more secure mobile phones.

Throughout our analysis for appropriate security policies and policy enforcement infrastructure for securing mobile phones, we perform our study under the following thesis statement:

*Mobile phones in the evolving mobile ecosystem can only be secured by identifying and addressing the stakeholders security requirements at the platform, application and content layers.*

We do not attempt to bring forth a complete security solution or to enumerate all possible security requirements of every stakeholders for all of the three perspectives. While it is viable to show the required conditions for a secure system, the complete set of sufficient conditions is unknowable. Instead, our research discusses several classes of conditions in the three basic domains that must be fulfilled to protect the important stakeholders in the mobile ecosystem through policy implementation. We illustrate how such infrastructure can be realized, the involved issues, and engineering challenges.

The Android platform is used as the vehicle for our study because it well aligns with the trend of convergence and openness of the mobile ecosystem. Nevertheless, the approach and the design principle is applicable for general mobile phone platforms.

We aspire to a flexible and extensible solution because the mobile phone usage will
always change in accordance with the gradual growth of the smartphone technology, the mobile phone applications, and the available content and services. Undoubtedly, we are facing a continually changing threat model. While this multi-cornered approach is believed to be valid over time, the practical solutions yet need to be adapted to such evolution.

1.2 Contributions

This work makes the following contributions:

- **Develop a methodology for identifying the stakeholders’ security requirements:** Although the security requirements engineering research has been widely studied in software and system development process particularly for enterprise-level software security, we have not witnessed such practice in the domain of mobile phone security. Enterprise and cross-enterprise environment is greatly different from mobile phone environment. Our research deliberates over the nature of mobile ecosystem and develops a three-prong methodology to retrieve the stakeholders’ security requirements for the mobile phone platform, mobile phone applications, and mobile content.

- **Identify security policies codifying the security requirements:** The characteristics of the mobile ecosystem do not support the inclusion of the stakeholders’ security requirements during the design of the mobile phones. To allow late integration, we develop security policy structure that is semantically rich enough to express such requirements to be imposed during runtime, taking into account the limitation of the environment under which the enforcement will be performed. We demonstrate the application of our methodology and create the initial sets of security policies which are appropriate for securing the mobile phone platform, the mobile phone applications, and the mobile content.

- **Provide appropriate infrastructure for securing the key elements of mobile phone activities:** To address the phones’ limitations, we design and develop the security infrastructures that accept the security policies from the stakeholders and perform policy enforcement such that the mobile phone platform, the mobile phone applications, and the mobile content are protected according to the policies.
1.3 Dissertation Outline

The goal of this dissertation is to seek for a pertinent scheme to secure mobile phones in the currently evolving mobile ecosystem. To achieve this objective, we study the mobile phone environment for an approach to devise the stakeholders’ security requirements and to develop appropriate security infrastructures that realize them to approach mobile phone security from the perspective of the mobile phone platform, the mobile phone applications, and the mobile content.

In Chapter 2, we provide a comprehensive description of mobile ecosystem. We examine how this immense network is recently evolving and the significance of such movement to the mobile phone security model. Following the roles of the major stakeholders, the chapter also presents the facts and figures that reflect their security requirements and expectations on the behaviors of the mobile phones. This chapter also explores the breadth and depth of the current state of the art in mobile phone security and analyzes the limitations of the current infrastructures in the presence of the recent evolution. In particular, it presents a number of contemporary attacks on mobile phones, from malware to botnets, along with a broad spectrum of related research and industry efforts, from application certification to penetration testing, as the attempts to mitigate such attacks. Later in the chapter, we provide essential background knowledge on the Android operating system which is used to demonstrate our approach to security. The chapter ends with an overview of security requirements engineering research discipline necessary to develop a full understanding of our avenue of research.

Chapter 3 introduces our approach to security from the angle of the mobile phone platform. More specifically, we reveal how open mobile phone environment drives malware problems. In response, we propose an on-platform application certification to protect the phone platform from malware according to the security requirements of the users and the platform developers.

Application convergence has encouraged mobile phone applications to expose their data and internal functionality to other applications as well as utilize those provided by others. Chapter 4 discusses how the trend of application interactions has taken down the protection shield around each application and pushed the application developers to protect themselves and their clients. The methodology for developing application security policies is presented in this chapter together with the required policy enforcement mechanism.

The rapid growth of cellular data traffic depicts the increasing amount of external content being delivered the phones. Chapter 6 demonstrates the need of the external
content and service providers to protect such content, ensuring that it is delivered to the designated phones, received only by the target applications, and handled only in the intended ways. Via the analysis of the real-world applications, this chapter presents the initial set of content security policies and our proposed policy enforcement infrastructure.

Finally, Chapter 6 draws the dissertation to conclusion by reviewing the advancement of the mobile phone security paradigm when incorporated with our security solutions. We end our discussion with the future of the research in this area.
Chapter 2

Preliminaries and Related Work

Apprehending mobile phone vulnerabilities and understanding our strategies for addressing the underlying security challenges require extensive background knowledge and a significant appreciation of a broad spectrum of related research work. The mobile phone system consists of various groups of stakeholders. Playing different roles in the mobile ecosystem, these stakeholders usually have diverse sets of requirements. This system also relies on different software, hardware, and networking technologies, some of which already engage with certain security models. As a result, the comprehension of these elements, their relationship, their design and operating constraints, their security ramifications, as well as the engineering techniques for soliciting the stakeholders’ needs is necessary for retrofitting and enhancing mobile phone security.

This chapter aims to introduce the key players and the important infrastructure components of the mobile phone environment and investigate the influence of the technological advancement on these elements. The chapter also analyzes how the existing machinery of the mobile phone system contributes, both positively and negatively, to security. Along with such discussion, we point out the concepts and techniques that can be applied to improve the security state of mobile phones. We start with the overview on the mobile ecosystem and consider its recent transformation which impacts the security of the system. Next, we go over the rudiments of the Android operating system, which is used in our study, and the basic mobile phone functionality relevant to our area of interest. Then, we describe some modern attacks on mobile phones together with the existing mitigation techniques. Later in the chapter, the security requirement engineering research principle to be used to understand the needs of the stakeholders is introduced. We bring out the necessity for a variant approach for acquiring and analyzing security
requirements in the mobile phone paradigm. Finally, we discuss the line of research in policy systems.

The information provided in this chapter will be referred to in the later chapters as we deliberate over the pros and cons of the available alternative solutions for the problem being considered. More importantly, it will also be used to show the rationale behind our design decisions.

2.1 Mobile Ecosystem and its Evolution

Mobile phones are considered secure only if they behave within the expectations of the stakeholders in the mobile ecosystem. On the other hand, the activities performed by these stakeholders also substantially affect the operations of the mobile phones. Given this relationship, the understanding of the mobile ecosystem, the stakeholders, as well as their activities is a vital preliminary to secure mobile phones.

The mobile ecosystem is comprised of multiple classes of members. They can be categorized in six major groups [8, 11] as shown in Figure 2.1. The first group, group (A), consists of the stakeholders forming the technological foundation such as the network operators, the infrastructure providers, and the hardware manufacturers. The backend parties like the service and billing providers are also included in this group. They establish essential hardware, software, network, and core services for the system. Group (B) consists of the platform providers. They develop the platform for both voice services and applications, which are created by the applications and software providers in group (C). Today’s platform also exposes many of the phone’s core functionality to the applications such as SMS/MMS, voice call, and location service. In group (D), the content and service providers offer different products such as games, media, and social networking services. Various government agencies as presented in group (E) also have important roles by placing the regulations to control the operation of the mobile ecosystem. Lastly, the consumers presented in group (F) include the phone users and the phone owners. It is their demands that drive this competitive market.

2.1.1 Evolution of the Mobile Ecosystem

The mobile ecosystem is progressively changing. With technology convergence and recent initiatives from different stakeholders in this mobile landscape, this ecosystem becomes more open and converging.

In the past, the mobile telecommunication industry was led by the network opera-
The mobile phone applications and mobile services were constrained by the device’s specification and the technical supports from the network operators. However, being a closed ecosystem with few dominators limits innovations thus can be destructive for an emerging business like the mobile telecommunication industry [8]. As a result, since 2006, we witness the gradual transformation of the mobile ecosystem as more stakeholders follow the principle of opening up their services [12]. Google launched mobile clients for Gmail [13]. Three UK mobile and T-mobile UK started to offer unrestricted and unlimited Internet access [14, 15]. Moreover, Three UK mobile also opened its network for Skype-to-Skype free calls [16, 17]. The trend of openness can be observed from several phenomena [12]. First, there are an increasing number of unencumbered applications and services such as Google Maps, Yahoo!Go, Skype, IM clients, and social networking services. Second, multiple-device/network platforms have become available, allowing the developers and the content providers to reach the mobile phone users directly. These open source mobile phone operating systems are designed to support cross-platform applications and online services, such as Android, LiMo, Symbian foundation, and Access Linux Platform (ALP). The publicly available software development kits also promote the development of innovative applications. Third, the network operators start to respond to the consumers and the competitive pressure. They increasingly allow external applications and devices to access their core network functions [18, 19]. Lastly, the devices themselves have become more powerful and include more features to support rich applications and services. The devices with open architecture also enable the developers to create tightly integrated applications.
In addition to becoming more open, the mobile ecosystem is also converging to provide anytime, anywhere, any device service accesses. Network convergence allows the integration of services residing in different networks such as analog telephone network, ISDN, IP phone networks, the Internet, and 2G/3G mobile cellular network. Terminal convergence, represented by smartphones, integrates multiple functions into a single mobile phone. Industry convergence brings together companies in traditionally separate industries as part of this mobile ecosystem. They collaborate and integrate their products, namely service convergences, to offer compelling services and gain some segments of this profitable market.

2.1.2 An Outlook for Mobile Phone Security

The changing face of the mobile ecosystem unavoidably impacts the threat and trust models as well as the assumptions on which the mobile phone security model base. We now consider the security implications of such openness and convergence.

In open application development model, mobile phone applications are no longer controlled by the network and platform providers. In particular, as the application development kits have become publicly available, there are an increased number of applications provided by the third-party application developers and amateurs. This model opens the opportunity for malware which has become a serious threat against mobile phones as will be described in Section 2.4.2. To make the situation worse, the availability of open unlocked mobile devices further increases the chance of installing malware because the users are allowed to install any applications and platform updates including those from unknown sources.

Furthermore, the invention of feature-rich highly integrated devices introduces a new class of attacks that goes beyond the application’s boundaries. Cross-service attack is a consequence of the convergence of computing and telephony. It leverages the interaction between different applications and services. Such attack has been shown to be successful on Windows Mobile Pocket PC [20] and iPhone [21]. While this type of exploitation can be contained by running applications in isolation, the trend of cross-applications communication in the current mobile ecosystem has brought down such protection boundaries.

As the mobile ecosystem moves towards service and industry convergence, more business domains begin to offer mobile services such as mobile and SMS-based banking, mobile and SMS ticketing, push services including push emails, and MMS marketing. However, as the inherent mobile information handling is still subjected to different types of attacks [22, 23], the introduction of such mobile services places these sensitive trans-
actions in jeopardy.

A more important consequence of the openness and convergence is their potential to aggravate the problems [24]. As shown in the cross-service attack example above, application and service convergence allows the vulnerabilities in one application to impact others. Recently, cross-application scripting attack on desktop environment has been revealed [25]. As smartphones start to move towards the same direction, it is highly likely that this type of attack will soon reach mobile phones. In a similar way, the capability of the mobile phones to exchange information with other phones or computing devices opens more channels to attacks and leads to more infection vectors, such as through file and application sharing, Bluetooth connection, and data synchronization. More seriously, the impacts of the compromised phones can potentially propagate to the network. The exposure of the core network functionality inevitably increases the attack interfaces and puts the network at risk. As an example, connecting SMS service to the Internet domain simplifies DoS attack which would deny legitimate voice and SMS service [26]. It has also been demonstrated that the attack on high-bandwidth cellular data services is feasible with a very small amount of traffic [27]. The work explained that it was the architecture of the network itself that was responsible for enabling such attacks.

In the existence of various attacks, the stakeholders have different security interests aiming at protecting themselves. The stakeholders forming the technological foundation and the backend providers in group (A) are interested in maintaining the availability and stability of the cellular infrastructure. They expect the phone platform to run as normal and be able to protect itself from malicious applications that may subvert the operation of the phone and in turn of the network such as launching denial-of-service attacks. The device manufacturers also rely upon the integrity of the device platform running on the hardware usually with access to low-level functionality. They expect that the messages they sent to the mobile phone, commonly through OTA interface, are well protected and will not be received by malicious applications.

As all the applications and native functionality depend on the operation of the device platform, the main concern of the platform providers, represented in group (B), is to protect the integrity of the platform and ensure that it operate as designed. They are interested in guarding the platform against malicious applications and inappropriate use of the core functionality. The platform must provide good support for the applications. For example, it must deliver the content it receives from the network only to the pertinent applications.

The application providers in group (C) usually concern about protecting their appli-
cations against attacks and misuses. They expect the applications to run as designed after they are installed on the phone. In particular, today’s mobile phone applications sometimes interact with other applications which may come from different providers. The applications start to expose their functionality and at the same time utilize features from other applications. With this model, the application providers are interested in ensuring that each interaction occurs in an expected way to protect their applications from other potentially dangerous software. When an application counts on the content from an external entity, the provider expects the content from the legitimate source and is delivered in a means that preserves confidentiality, integrity, and privacy.

In group (D), the content and service providers being connected from the mobile phone also have some confidence in the phone. Generally, they expect that the device platform and applications will not misuse their content and services, and that the content delivered to the platform will be distributed in an agreed way.

The government agencies in group (E) are interested in the security of the mobile communications as a whole. They watch over all of the stakeholders in the mobile ecosystem to ensure that their operations abide to the regulations termed by the government. Their concerns over the mobile phones include the protection of the device platform, the applications running on them, as well as the content stored on and delivered to the phone. Finally, the consumers (F) currently place increased trust on their mobile phones. Apparently, the users expect the phone platform and the applications they use to protect their personal transactions and stored data. They also expect the content exchanged with external parties to be properly handled.

2.2 Android Operating System

We use Android mobile phone operating system as a vehicle to demonstrate our approach to mobile phone security. Being an open source platform with publicly available application development kit, support for data sharing and application integration, as well as advanced and extensible features, Android is a good representation of smartphone platforms in the modern mobile ecosystem. This chapter provides an in-depth discussion of the Android operating system, particularly its security model and its limitation in fulfilling the stakeholders’ security requirements.

Android is a smartphone platform developed by Open Handset Alliance (OHA) [28] led by Google. Since its announcement in November 2007, Android has garnered a lot of public attention. In addition to the official releases of Android and the updates
provided by the Android technical team\(^1\), several Android community ROM are also available \([29, 30]\).

### 2.2.1 Android Architecture

Android can be described as a software stack that includes an operating system, middleware, and key applications \([31]\) as shown in Figure 2.2. The main part of Android is the middleware which runs on the top of the Linux Kernel V.2.6 and the hardware drivers. It comprises of the libraries, the runtime environment, and the application framework which offers necessary components and services to the applications. Lastly, at the top of the stack are the applications which access the underlying functionality through the APIs and communicate with one another via its unique lightweight IPC mechanism.

In Android, all functionality including the core ones (e.g. Phone, dialer, Contact book and Messaging functionality) is implemented as applications and treated in the same way as the third-party applications. To isolate applications from each other, processes of the same application run with the same unique UNIX user ID.

There are two groups of applications in Android. **System** applications are bundled with the platform and stored in the read-only system partition on the phone, while **user** applications are obtained from the Android Market or other sources and installed by the users. Each application is distributed as an Android application package (.apk) which includes the application’s configuration file called *manifest* as outlined in Figure 2.3. This file describes the application’s structure, namely its components, as discussed in Section 2.2.2. It also includes three pieces of security-related information, namely the permissions it declares, the permissions it requests for, and the permissions protecting its components, as detailed in Section 2.2.3.

### 2.2.2 Android Application Model

Android’s application model is highly flexible. An application is composed of one or more components. A component may be invoked as a separate process or under an existing process in the same application. Four types of components are defined in Android:

*Activity* components are application’s user interface. They interact with the users via touch screen and keypad. Each screen presented to a user is usually a separate Activity. While there are normally multiple activities in the system, only one of them can be active at a time.

\(^1\)The latest release at the time of writing is Android 2.1 r.1 released in January 2010
Figure 2.2. Android architecture consists of three layers: Linux kernel, Android middleware, and applications.

Figure 2.3. Android’s application manifest file contains the application’s security configuration and the component structure.

Service components are background processes that continue even after the applications lose focus; for example, to download content or play music. They can export Remote Procedure Call (RPC) interfaces including support for callbacks to interact with other components.

Content Provider components offer functionality similar to relational databases. They store and share data with other components. While the Content Provider APIs are SQL-like interface, namely select, update, insert, and delete, the backend implementation is left to the application developer. Content Providers are addressed via unique Content URIs which are registered with the system at install time.

Broadcast Receiver components provide a generalized mechanism for asynchronous event notifications. They listen to and receive message objects called Intents from other
components in the same or different applications.

Components within the same or from different applications can interact through the Binder IPC mechanism. Intents provide a general means for components to interact with one another. They are used to start Activities, start Services, and send messages to Broadcast Receivers. An Intent includes two main pieces of information; namely, the address of the receiving component and the data to be passed to such component. The sender of the Intent may explicitly specify the receiving component by name. A more prevailing manner is to implicitly address the Intent with action string describing what action has occurred or what action is to be done. The action string can be one of the Android's predefined actions such as VIEW, DIAL, and SMS_RECEIVED, or a developer-defined action string. The applications wishing to receive any certain Intents declare their assertion with the system by defining the matching Intent Filters. When the Intents are sent out, the Android middleware automatically resolves for the appropriate components for handling such event based on the declared assertion.

2.2.3 Android Security Model

Built on the top of the Linux kernel, system-level security of Android is provided by the underlying Linux internals. Linux security enables strong user-level isolation by running each application in a separate Dalvik virtual machine with a unique user identity.

A more promising security framework is provided at the middleware level to govern its application-level IPC mechanism which cannot be mediated by the underlying Linux system [32]. The Android middleware mediates the interaction among components and enforces mandatory access control (MAC). The access control policies are based on permission labels, each of which represents a unique permission.

Applications can declare new permissions which are normally used to protect their own components. They specify the permission labels and the corresponding protection levels in the manifest files. Permissions with Normal protection level are granted to any application that request for them. Dangerous permissions require user confirmation. Signature permissions are granted only if the application is signed using the same developer key as the one used to sign the application that declared the permission. Lastly, signature or system permissions are granted if the application is eligible for the signature permission or if it is in the Android system image or is signed with the same developer key as those in the system image.

For an application to acquire a particular permission, it specifies its request in its manifest file. The permission request is examined by the system at the application’s
installation time. Then, the permission is granted as allowed by its protection level.

Mainly, permission labels are used to protect application components and certain library APIs used to access system resources such as Internet access, user’s location, camera, and phone call operation. Each of the Activity, Service, and Broadcast Receive components can have only one permission to protect it. In contrast, a Content Provider component can be protected by two permissions, one for read access (i.e. query operations) and the other for write access (i.e. insert, update, and delete operations). When a component initiates access to another component protected by a permission, the middleware intercepts the IPC. The access is allowed if the application in which the calling component resides possesses such permission. Figure 2.4 from [32] demonstrates the IPC’s access control mechanism.

![Figure 2.4](image.png)

**Figure 2.4.** Android middleware mediates access between components and enforces mandatory access control (MAC) policy. As application 1 possesses label $l_1$, component A in application 1 can access component B protected by label $l_1$. However, it cannot access component C assigned with label $l_2$.

Identical to the way the components are protected, Intents can also be protected with permissions to allow only the components assigned with such permissions to receive them. However, unlike components whose protection permission labels are specified in the manifest, the permission labels protecting the Intents are defined within the source code, making security analysis more difficult.

A drawback from allowing only one permission to protect each Service component involves the granularity of the access control. The possession of the permission enables all operations that can be performed on the Service, namely bind, start, and stop the service. To handle this issue, the Android framework offers `checkPermission` method to enforce additional security check including regulating the call to each method. However, such access check is invisible for security analysis without source code inspection.

Android also offers another means of communication via **Pending Intents**. They are reference objects to be handed to other applications to allow the receiving applications to perform certain actions on behalf of the applications that create them. Along with the receiver’s address and the data, a Pending Intent also contains an internal Intent. The receiving application can add the data to the internal Intent and invoke the associated
action as described by the internal Intent. A similar mechanism is URI permission. It lets one component delegate to another component read or write permission on a certain data record identified by URI. Clearly, both mechanisms involve delegation. While they provide more functionality or in some cases more security, delegation may complicate policy evaluation.

2.2.4 Limitations of Android

The Android’s security model is deeply scientific especially when compared with other existing mobile phone operating systems. Nevertheless, it is still semantically limited and does not give a sufficient foundation for the level of security we want to achieve.

At high-level, Android is also subjected to the two challenges introduced in Chapter 1. First, like other smartphone operating systems, Android does not carry a clear notion of security, partially due to the absence of the standard usage model or the functional requirements in smartphones. Instead, the usage model usually depends on the features packed with the phone themselves. Therefore, framing the properties of secure phones including the Android-powered phones is still an open problem.

Security model is an abstraction of security policies [10]. By taking security policies from the stakeholders at a very limited extent, the security models of most of the currently available mobile phone platforms are created and verified mostly by the platform developers with only small input from important stakeholders. The security model of Android is designed by its developer, the Open Handset Alliance, incorporating only limited security requirements from other stakeholders. OHA is a group of 65 technology and mobile players from all parts of the mobile ecosystem, ranging from network operators, hardware manufacturers, software vendors, to commercialization companies. Unfortunately, the contribution that these members have on the Android platform is for the most part related to its functionality. Most of them work with OHA to create Android-compatible devices, software and hardware products, and services. Some members also contribute their intellectual properties to the OHA, in terms of code or libraries to the project. For instance, PacketVideo delivered its libraries to Android’s Media Framework [33], while SVOX has brought its speech output solution to Android [34]. The platform security itself is developed by the Core Technical Team [35] assisted by two security consultant companies, iSEC partner Inc. [36] and n.runs AG [37].

Since the major stakeholders of the mobile ecosystem barely contribute to Android security, the degree to which the resulting security model reflects their security requirements is unidentifiable. In reverse, their ability to protect themselves are constrained by
the security solution designed by the platform developer.

Nevertheless, we have pointed out that it is the characteristics of the mobile phone environment that impede the inclusion of most of the stakeholders’ security requirements into the operating system security design. Still, the security model should allow the accommodation of these requirements at the later time. While Android accepts the application policies as part of the applications’ manifest files, these straight permission-based policies are static and very coarse-grained. For instance, the permissions are used to control total access to the resources. The access is fully allowed with permission and completely prohibited without permission. Furthermore, they lack in semantics and expression power which are required to describe certain classes of the stakeholders’ security requirements. Accordingly, the current Android policy enforcement mechanism is simply founded on permission matching. While this mechanism is sufficient for its currently supported policies, it cannot handle other types of security policies that the stakeholders may want to impose.

Throughout this dissertation, we address such limitations of Android. We equip Android with the capability to support a wider range of security policies in response to the stakeholders’ security requirements through the implementation of the platform policy, the application policy, and the content policy, as well as their corresponding enforcement infrastructures.

### 2.3 Basic Mobile Phone Operations

In general, smartphones support two classes of functionality, voice service and data service. They operate using two processors. The application processor performs tradition operating system tasks such as memory management and executes user applications including camera, games, and music player. The second CPU is the baseband processor which runs a specialized real time operating system. To allow the applications to use the network, the telephony software stack on the application processor and the modem on the baseband processor communicate by passing text-based commands through some serial lines. In most mobile phones including Android-based phones, the modem is controlled by the GSM Attention, or AT commands [38].

All the network-related functions are handled by the telephony software stack. On Android, this component is comprised of the Radio Interface Layer (RIL) daemon which communicates with the modem. The RIL daemon interacts with the Android’s Phone application, namely `com.android.phone`, running above it. Mainly, it is the Phone
application that handles all the cellular network-related operations, for both voice service and cellular data service. Our research pays special attention to data service which has recently become more prevailing and turned to be an attractive target for the attacks.

### 2.3.1 Short Messaging Service (SMS)

The most primary cellular data service is short message service. The SMS network provides an infrastructure for delivering short messages to and from mobile clients or computing systems. As shown in Figure 2.5, the heart of the SMS system is the Short Message Service Center (SMSC) which acts as a store-and-forward system for short messages [39]. It receives the short messages from the mobile clients inside the cellular network and from the External Short Messaging Entities (ESMEs) such as web-based SMS portals, voice mail system (VMS), and email notification system. When a short message arrives, the SMSC determines the content, transforms it to SMS format, and place it into the SMSC queue. The cellular network infrastructure finds the destination mobile client and routes the message to the client.

![Figure 2.5. Simplified SMS network architecture](image)

The SMS messages are delivered through a control channel of the network. The network usually guarantees SMS delivery. If the receiving mobile device is unreachable, the SMSC will store the short message until it becomes available. While they are encrypted in transit by the network providers, there is no end-to-end (device-to-device) security. The messages are encrypted with the key shared between the devices and the network provider, not between the end devices.

### 2.3.2 Multimedia Messaging Service (MMS)

MMS is a system which allows the mobile clients to perform messaging operations with a variety of media types. In addition to enabling messaging among mobile clients, it also supports the messaging activities with other systems including email system and legacy messaging system such as paging and SMS.
On the mobile client side, the MMS user agent is responsible for composing and rendering multimedia messages using the appropriate content rendering service on the platform. It also sends and receives multimedia messages using the message transfer service of the suitable protocol.

MMS system centers on Multimedia Message Service Center (MMSC) [40]. In normal setting, the MMSC consists of MMS server and MMS proxy relay as shown in Figure 2.6 and 2.7. The MMS server offers storage services for multimedia messages, while the MMS proxy relay is responsible for the messaging activities and providing access to the MMS server. Nevertheless, the mobile clients do not communicate with the MMSC directly but through the WAP gateway push proxy.

The most common use case for MMS is client-to-client messaging. In this scenario, the sender composes and addresses the message. As presented in Figure 2.6, the message is submitted to the client’s associated MMS proxy relay as defined in the MMSC settings of the phones [41]. The message goes through the WAP gateway push proxy to the MMS proxy relay. The MMS proxy relay resolves the target address. If the receiver is on the same network, it stores the message in the MMS server. Otherwise, the message is forwarded to the target’s MMS proxy relay and will be stored on the target network’s MMS server. After that, the sender’s MMS proxy relay sends the send confirmation message informing the success or failure back to the sender. On the receiving side, the receiving MMS proxy relay notifies the receiving client using a WAP push message M-Notification.ind over SMS as shown in Figure 2.7. In response, the client creates TCP/IP connection to retrieve the message from the MMSC through the WAP gateway push proxy. After all, the receiving client sends the acknowledgment indicator to its MMS proxy relay. If the sender is in a different network, this indicator is forwarded to the sender’s MMS proxy relay and finally passed to the sending client.

In Android, the incoming SMS, MMS and WAP Push messages arrive at the RIL daemon which communicates with the Phone application. After determining the type of the messages, validating their structures, and processing them, the Phone application distributes the messages to other applications that have registered to receive those particular types of messages.

Android applications can assert their desire to receive the incoming SMS and MMS by registering the Intent Filter for the action string SMS_RECEIVED and WAP_PUSH_RECEIVED with the system with respectively. They must also possess RECEIVE_SMS permission to receive SMS and RECEIVE_MMS to receive MMS.
2.3.3 Internet Connection

Mostly, smartphones can connect to the Internet either through cellular data network using their radio interface or through WLAN using their Wi-Fi component.

To exchange data packets with external networks via the cellular data network, the mobile phone needs to establish a Packet Data Protocol (PDP) connection with the network via the Radio Interface Layer. In contrast, connection with WLAN is performed through the WiFi driver interface. The switching between cellular data connection and WiFi connection is automatically handled by the 3G adapter transparent to the applications. To access Internet, the applications use the tradition Java network API. Android network API is also available for addition networking functionality. In any case, the applications must hold `INTERNET` permission to create the Internet socket connection.

2.4 Attacks on Mobile Phones

Even though cellular networks as well as modern smartphones usually encompass some security mechanisms, it is evidenced that these technologies still have a lot of vulnerabilities. The incorporation of advanced services which is a result of the mobile ecosystem evolution introduces even more attacks and unforeseen consequences.
2.4.1 Cellular Network Vulnerabilities

Historically, the isolation of cellular networks from the external entities has at a great extent provided security advantages. The cryptographic algorithms employed within these networks are proprietary and have been kept secret. However, as affirmed in Kerckhoffs’ law and widely accepted by cryptographers, a strong cryptosystem must be secure even if everything about the system except the cryptographic key is public knowledge. By relying on the secrecy of the system design and implementation to provide security, these networks have eventually exhibited vulnerabilities.

Subscriber identity authentication is one of the most critical security functions in GSM networks. It is performed using A3/A8 algorithms. While these algorithms are network operator dependent, it is publicly known that the majority of the network providers employ some variants of COMP128 algorithm. When this private authentication algorithm is leaked to the public in 1998, it has been proved to contain weaknesses which enable GSM cloning if the adversaries have physical access to the phones [42]. The tools and techniques have subsequently become widely available.

GSM networks provide confidentiality using encryption for voice data and signaling data over the radio interface. The A5-famility algorithms are used to provide this functionality. The A5/1 algorithm provides a stronger security guarantee that the A5/2 algorithm, while the A5/3 algorithm is not currently implemented. The A/2 algorithm has been shown to be susceptible to cipher-text only attack, which allows the attacker to recover the encryption key within a few milliseconds [43]. This attack can also be extended to an active attack on the A5/1 algorithm. Recently, a group of Black Hat researchers have created an inexpensive and simple FPGA-based tool to directly compromise the A5/1 encryption and successfully intercept voice conversations and SMS text messages within 30 minutes [22]. Undeniably, compromising the GSM’s cryptosystem has now become significantly more practical.

2.4.2 Mobile Phone Vulnerabilities

As billions of mobile phones have been brought to the mainstream of people across all socioeconomic classes and of almost all businesses sectors, they have unavoidably received unwanted attention from attackers, ranging from hobbyists to criminals.

Malware is still the most dominant threats against mobile phones. The first real-world mobile phone malware called Cabir [44] was discovered in 2004 as a proof-of-concept worm which simply uses Bluetooth as its propagation vector. Based on Cabir, Lasco [45]
additionally infects all available software package (SIS) files residing on the phone on the assumption that the user might share them. Commwarrior [46] added MMS propagation in addition to Bluetooth. These early generation malware target Symbian OS and do not exploit any vulnerabilities. The Skulls [47] Trojan provided one of the first destructive payloads, also targeting Symbian OS. It overrides all applications with non-functioning versions, rendering all applications useless. The Cardblock [48] Trojan embedded within a pirated copy of InstantSis (a utility to extract SIS software packages) sets a random password on the phone’s removable memory card, disallowing access to the user’s data. A Trojan horse named SymbOS.Locknut [49] can crash Symbian V.7.0s devices.

As other mobile phone operating systems became widely used, we started to witness malware targeting them. A Trojan program called Palm.Liberty could delete all applications on Palm OS devices and reboot the system [50]. Brador opens the backdoor for the attackers to upload or download files from Pocket PC devices and to run arbitrary commands [51, 52]. RedBrowser [53] which is the first J2ME phone Trojan comes in form of J2ME WAP browser.

Another family of mobile phone malware is Spyphone software, such as MobileSpy [54], Flexispy [55], and Neo Call [56], exists for various OSes. After it is installed on the target phone, the spyware sends call history, incoming and outgoing SMS messages, data logs, and Bluetooth usage, among others, to the software vendors server via GPRS or SMS. Then, the customer of the spyware service can logon to the server to monitor the target phones activities.

 Mostly, such malware relies on the users to allowing the installation of them, such as by Internet downloading and application sharing. Recently, a Blackhat researcher can send spoofed SMS, which bypasses the network operator, to deceive the users to install malware [57].

Cross-service attacks, a less common yet highly destructive class of attack, are increasingly more prevalent. In this type of attack, the adversaries exploit vulnerabilities within a service or some functionality, such as buffer overflow vulnerabilities and other software vulnerabilities in input parsing, to abuse other services or functions. In the previous days, these exploits were major problems only in computer systems. However, it was only a matter of time before they move to smartphones.

In spite of the fact that the cellular infrastructure already provides in-network sanitization for SMS and MMS, a security analysis of PocketPC MMS User Agent through fuzzing technique has revealed a number of buffer overflow vulnerabilities [58]. Some of them enable the manipulation of the program counter of the application’s process. More
severely, some vulnerabilities allow the adversaries to remotely inject arbitrary code on the target device using an MMS message.

A different fuzzing technique is applied to perform vulnerability analysis on iPhone, Windows Mobile, and Android based phones against malformed SMS messages [38]. The study illustrates the injection of SMS messages below the telephony software stack to test all SMS-based services without the need of actual incoming messages from the network. Again, the work reveals several security holes. For example, faulty SMS messages can crash iPhones SpringBoard or disconnect both iPhone and Android phone from the network.

In the same vein, SMS application on iPhone was found vulnerable, which allows an attacker to remotely install and run unsigned software code sent by SMS with root access to the phone [59]. Moreover, iPhone’s web browser was also shown to contain vulnerabilities [21]. A different work exploited buffer overflow vulnerabilities in FTP server process for Windows CE [20]. Though a proof-of-concept system, it was shown that by injecting malicious payload, the adversaries would be able to access the cell phone functionality and some other cellular-based services.

2.5 Attack Mitigations

In response to the rise of attacks against mobile phones, the mobile phone industry and security researchers endeavor to mitigate the problems, limit their catastrophic effects, and prevent new classes of attacks.

While mobile phone malware thus far are not as debilitating as computer malware, most security experts agree that they are on the brink of their destructive power. Since majority of them show up disguised as applications, a promising strategy for malware mitigation is to prevent the users from installing them. However, the users cannot foresee the behaviors of the applications before they are installed. As a result, an effective approach is to have the application certified by a trusted evaluator. The Application Security Framework [60] offered by the Open Mobile Terminal Platform (OMTP) recommends a certificate-based mechanism to determine the applications’ access rights based on their origin. A good example is signed MIDlet implementation. The MIDP 2.0’s security follows the concept of protection domains which are defined by MIDP implementors (e.g. the manufacturers and the network operators) to control the use of the sensitive operations such as network access, file systems access, and personal information management [61]. It adopts the code signing scheme which allows the application
developers to pre-establish trust with the MIDP implementators. At install time, the MIDP implementation on the device verifies the developer’s signature, the corresponding certificate, the intermediate certificates (if any), and the hash of the application, which are included in the application descriptor. Based on the developer’s identity, the MIDlet will be placed in the appropriate protection domain.

Modern smartphones promote the distribution of applications through application market places or directories such as Apple’s App Store [62], Symbian Horizon [63], and Blackberry’s App World [64]. This distribution model allows a more rigorous malware mitigation regimen which is to vouch for the applications. For instance, Apple and Blackberry require all applications to be reviewed before being placed on the market places. All applications must be Symbian Signed [65] to be admitted to the Symbian Horizon directory. The Symbian’s Symbian Signed program is adopted among Symbian-based mobile phone licensees. The handsets with properly installed certificate are able to recognize Symbian-signed applications. They display warning when an unsigned application is about to be installed as well as prevent unsigned applications from accessing the protected interfaces. Nevertheless, while application vouching is a widely accepted approach, there was still some evidence that malicious applications could pass the App Store’s security test [66, 67] and received proper signature.

Anti-virus software provides a second layer of defense against malware. F-Secure (www.f-secure.com) is one of many security solution providers for Symbian and Windows Mobile. However, like PC anti-virus software, protection is reactive and depends on updated virus signatures. Behavior signatures [68] considering temporal patterns of malicious behavior also show promise and may defend against malware variants with different binary signatures. Similar to behavior signatures, multiple network-based anomaly detection systems have been proposed [69, 70]. These systems report phone activity (e.g., SMS and Bluetooth usage) and runtime features (e.g., CPU and memory usage) to a central server that performs statistical anomaly analysis to detect mobile phone malware epidemics.

The discovery of cross-service attacks has alarmed the application developers to patch vulnerable applications and on-platform functionality. Nonetheless, while writing quality code with proper security testing is a good start for effective vulnerability mitigation, producing flawless software is still extremely impractical if not impossible. Therefore, a more achievable alternative is to limit the impact of the exploit. A labeling mechanism was proposed to monitor the processes’ system calls in Windows CE in order to contain any potential exploits within the service interfaces (e.g. WiFi and cellular interfaces)
from which the processes originate [20]. With this technique, the kernel reference monitor labels the executing codes based on their access to the service interfaces. Such labels are transferred back and forth between the processes and the system resources being accessed. When the processes want to perform any expensive operations, the labels are compared with the predefined policies to determine whether such operations are allowed. Another proposal suggested the adoption of Linux Security Module (LSM) as the hooking mechanism in LSM can be used to expand RBAC to detect cross-service attacks [71]. In this framework, the services and resources are defined as part of access control profiles in advance. The profiles are checked against before the service interfaces can access the resources.

Android operating system is designed to run each application in a separate Dalvik VM. As a result, it provides desirable isolation property which consequently limits the impact of any exploits. However, the IPC mechanism which allows interactions among application components as described in Section 2.2.2 inevitably opens channels for attack propagation. In Chapter 5, we will demonstrate our approach to mitigate such drawback through the adoption of application security policies.

Low-level security mechanisms such as Linux Security Module (LSM) framework and Trust Platform Module (TPM) have been accepted provide high degree of security to computer systems. Several research efforts bring forward the application of these technologies in mobile phone paradigm. They are usually proposed as the mechanisms to regulate the behaviors of the mobile phone applications during runtime. For instance, the trusted mobile phone reference architecture [72] realized the Trusted Mobile Phone specification using an isolation technique developed for mobile phone platforms.

Linux-based phones can also be secured by using the LSM framework to enforce different policies for platform protection. A compact SELinux security policies for Openmoko [73] was developed to justify the integrity of the phone and the trusted applications and to prove the phone's integrity to the remote parties. The scheme utilizes the Policy Reduced Integrity Measurement Architecture (PRIMA) [74] to measure the trusted code and information flows to ensure their integrity. When an application is installed, a MAC labeling service labels the installed files based on the predefined MAC policy. These labels are used by PRIMA during the application's runtime. To allow multiple stakeholders to control the operation of the applications, Rao et al. [75] developed a mandatory access control (MAC) system which consults the stakeholders in dynamically creating the policies for permission assignment. The application is contained in a soft sandbox by default. It can obtain the permissions to perform the operation outside the
sandbox if such permission granting is not prohibited by the policy and is authorized by the stakeholders. The access state of the application is maintained as execution role recognized by the stakeholders.

Windows Mobile .NET compact framework regulates the applications using security-by-contract [76]. Each application is attached with a behavioral profile (i.e. its contract) to be enforced at runtime. This method can be used to safely execute potentially malicious code [77].

As mobile phones increasingly process and maintain critical private and enterprise data, ranging from address book, calendar, emails, to financial information, another research direction which receives a lot of attention involves securing the data stored on the phone and the content delivered from the remote parties. The data-centric security model (DCSM) developed by IBM can be applied to select an appropriate scheme for protecting each piece of data stored on the mobile phone based on its value [78]. A more general protection technique entails the use of smartphone encryption products such as AlertBoot [79] and GuardianEdge [80]. Due to the lack of end-to-end security in the current message delivery between mobile phones and the existing vulnerabilities in the cellular networks as detailed in Section 2.4.1, there are several efforts for securing message delivery particularly in SMS communication. For instance, SMS 007 [81] is a security software to protect SMS communication using AES encryption in addition to those provided by the network operator. It is compatible with mobile phones that support Java MIDP-2.0. Pursuing a different approach, SMS-based VPN has been proposed [82] to secure SMS-based services based on PKI. A secure SMS message structured has also be proposed to include security-related information such as encryption information, sequence number, and message digest into the message [83].

2.6 Security Requirements Engineering (SRE)

A prime characteristic of the mobile ecosystem is the encompassment of a large number of stakeholders, each of which potentially has a diverse set of security interests. As emphasized in Chapter 1, the notion of secure mobile phones depends on the expectation from each of these stakeholders, which is expressed in terms of security policies.

A security policy defines secure for a system or a set of systems [10]. It is a statement of requirements that explicitly defines the security expectations to partition the states of the system into a set of secure (expected) states and a set of non-secure (unexpected) states. More specifically, a security policy is a set of policy statements which represent
the security requirements describing what behavior must take place or not take place in order to define the secure states of the system [10]. Our goal is to develop security policies for mobile phones by extracting the set of security requirements that different stakeholders have on the phones.

In comparison with system engineering and software engineering domain, a critical step in building a system is to determine the needs and the required conditions of various stakeholders of the system. As a extension to such concept, the area of security requirements engineering (SRE) provides different approaches to systematic analysis of the stakeholders’ security concerns in order to integrate security into the system design.

As security policies represent security requirements, we follow security requirements engineering practice to identify such policies. SRE builds on requirements engineering and security engineering. It involves the process of eliciting, modeling, and analyzing the stakeholders security requirements through the understanding of the concepts from these two areas: (i.) security-specific concepts from security engineering such as assets, threats, trust, and permissions, and (ii.) requirements elicitation concepts from requirements engineering, including interactions and dependencies among the entities in the systems.

In the domain of SRE, assets are both tangible (e.g. components and information of the systems) and intangible (e.g. capabilities, skills, and reputation) resources that have value. Functional requirements define how a system is supposed to operate in normal environment. Threats are potential attacks targeting the assets that may lead to harm. SRE aims to create security requirements which are the constraints or conditions that protect the assets against the threats.

Security Requirements Engineering has been widely adopted in software development process particularly for organization-level systems. The implementations have seen great success in practice. For example, Microsoft uses the Security Development Lifecycle (SDL) for the development of their software that must withstand attacks [84], and Oracle has developed Oracle Software Security Assurance (OSSA) as its secure software development process [85].

The initial efforts in this area consider security after the system design specification becomes available. Misuse cases [86, 87], Anti-Goals [88], and Anti-Models [89] view security requirements as the negation of the functional requirements. For instance, the misuse cases are derived from the legitimate use cases of the system. The potentially malicious entities are modeled as misusers with corresponding misuse cases. Then, the "threaten" relationships are drawn from the misuse cases to the use cases. By identifying unwanted action sequences of such misuse cases, this scheme can identify the security
requirements (i.e., security use cases) for avoiding such dangerous sequences of actions. Similarly, Anti-Models approach follows the concept of Anti-Goals to create a method for elaborating security requirements by building two models iteratively and concurrently. First, the system-to-be model is drawn to describe the software and its environment. Then, the Anti-Model is derived from the system-to-be model to illustrate how the specifications of the model elements could be threatened, why, and by whom. These early approaches indicate the set of behaviors that the system should exhibit and the threats that potentially impede them. However, they are not designed to express what should not be performed by the systems.

A different group of work in the same generation models the security requirements along with the conceptual model which captures the system’s functional requirements. SecureUML [90] uses UML to model access control policies and integrate them into the model-driven software development process. UMLSec [91] is another extension of UML to model the security-related features included in the system such as the cryptosystem and the access control mechanisms to represent the system’s security requirements.

The recent approaches integrate security into the software development process since the beginning. They are performed in parallel with the system development. For instance, System Quality Requirements Engineering (SQUARE) [92, 93] consists of two phases. The stakeholders’ inputs are collected in its first phase along with other preceding activities to acquire the agreed-on definitions, goals, and risks. Then, the second phase applies different requirements engineering techniques to elicit, categorize, and prioritize the security requirements. Security Requirements Engineering Process (SREP) [94, 95] extends SQUARE to include the Common Criteria (CC) and the notions of reuse. The Common Criteria is used as a formal method to validate that a particular product satisfies the defined set of security requirements. In similar fashion, Comprehensive Lightweight Application Security Process (CLASP) [96, 97] is comprised of a set of activity-driven, role-based process components. It embraces 24 main activities to be equipped into the software development process; seven of which capture the security requirements.

Another category of modern approaches suggests the inclusion of the context in which the system operates into the model. Secure Tropos [98, 99] is a derivative of i*/Tropos [100] which is an approach for organizational modeling. The Secure Tropos is designed for modeling and analyzing security requirements using the notions of ownership, trust, and delegation. It builds functional requirements model and then captures trust and privilege management requirements at organizational level to be included in the requirements analysis. By transforming to Datalog representation or Formal Tro-
pos, Secure Tropos allows formal verification of the consistency between the security and trust requirements.

Security Requirements Engineering Framework [101, 102, 103] is based on Problem Frames [104]. It uses context diagram and problem diagram to bound the context for each function of the system and to explain the problems. The functional requirements of and the assets associated with the function are determined. Its security requirements are developed as the constraints on the functional requirements. Requirements satisfaction argument is developed to assert that the system that follows the specification can satisfy the security requirements under the given set of trust assumptions.

Given the system’s security specification, threat modeling [105, 106, 107] tests and identifies the motivation and methods that the attackers could use to compromise the system. The threats for each object in the data flow can be discovered using STRIDE approach [108]. Moreover, attack trees [109, 110] can also be brought into play as a tool to determine the possible attacks paths. The use of attack trees enables the decomposition of the threats into a series of steps, which accordingly manifests them and reveals the vulnerabilities. The threat modeling technique is intended to be used during system design time and requires a lot of design specification information including dependencies, trust level detail, and component’s data flow diagrams.

Recently, security patterns for security requirements capturing have been developed. Based on the concept of design pattern, a pattern is a solution to a problem that arises within a specific context. When applied to security discipline, the security patterns method [111] amalgamates the extensive knowledge about security with the structure provided by the patterns. It aims to provide the guidelines for secure system construction and evaluation. Mostly, the current available security patterns for requirements elicitation involve the process of defining the assets, analyzing the reasons for protecting them, and finally specifying their security requirements [112, 113]. Some examples include the Security Needs Identification for Enterprise Asset pattern, the Threat Assessment pattern, and the Vulnerability Assessment pattern.

### 2.7 Policy Systems

A highly acknowledged design philosophy in computer science including computer security is the separation of mechanism and policy [114, 115]. Our research approach follow the same principle by using SRE to develop appropriate security policies which will be enforced by our proposed mobile phone security infrastructure. In this section, we provide
an overview of some well-known policy systems and policy enforcement techniques.

2.7.1 Classic Policy Models

One of the most fundamental security models involves the use of Access Control Matrix \[116\] to represent the system’s access control policy. It abstracts the protection state of the system as the set of objects and subjects. The matrix contains the rights that specify how the subjects are allowed to access the objects. The Access Control Matrix model is commonly implemented as Access Control Lists (ACLs) or as Capability Lists. The model is widely used especially to provide file system protection. Nevertheless, it does not give a complete description of the system’s access control policy. For instance, the model does not cover the rules regulating the modification of the access rights.

Role-Based Access Control (RBAC) \[117\] was proposed as a more appropriate access control model for non-military systems than Discretionary Access Control (DAC). The RBAC policy bases access control decisions on the roles individual users take on rather than their identities. The access permissions are associated with roles which are determined from the set of transactions that the users can perform within the context of an organization. The approach leads to more flexible and manageable system. The model is general enough to simulate other security models \[118\] and to model the environment for context-aware security \[119, 120, 121\].

Information flow policy is an immense area of research. It aims to control how information is allowed to move throughout a system. The Bell-LaPadula (BLP) model \[122, 123\] is the most classic information flow confidentiality policy for multi-level security (MLS). MLS is a lattice-based model \[124\]. The security levels are organized in a hierarchy which depicts the direction of the legitimate information flow \[123\]. The subjects are assigned with security clearances depicting their trustworthiness while objects have sensitivities associated with them. Based on such concept, BLP assigns security labels to the subjects and the objects in the system. Higher security labels indicates more sensitive information. Additionally, to allow access on “need to know” basis, the model supports security classification based on categories which describe the type of information. A security label together with a category forms a security level in BLP. The BLP model defines two rules. The simple security property allows a subject to read an object only if the subject’s clearance dominates the object’s security level (no read up). The *-property allows a subject to write to an object only if the object’s security level dominates the subject’s clearance (no write-down).

The matching model to BLP is the BIBA model \[125\] which offers integrity informa-
tion flow policy. It also defines two rules, dual to the BLP’s rules. The simple-integrity rule does not allow the subject to read from the objects with a lower integrity level. The *-integrity rule does not allow a subject to write to an object with higher integrity label.

2.7.2 Decentralized Policy

Most classic security policy models are designed to operate within a single security domain. The problem becomes more complicated in a highly decentralized environment such as the Internet which involves different security domains. To provide security in such settings, decentralized policy models have been developed.

A common adoption of decentralized policy system is in the area of trust management (TM), which is the problem of formulating security policies and security credentials, determining whether a particular set of credentials satisfies the relevant policies, and deferring trust to the third party [126]. It enables two systems with no preexisting relationship to establish trust and perform authorization before engaging in any particular transactions or allowing certain operations. As the identity of the participants (i.e., the subjects) may not be recognized across security domains, learning their identities may not be useful for determining their levels of trust. A more promising avenue is attribute-based authentication which involves the use of credentials describing the attributes of the subjects. Trust negotiation offers mutual authentication through bilateral exchange of credentials. Access control policies specify the required combinations of credentials for gaining access to the local resources or credentials [127, 128].

The early research on trust management centered on trust establishment using certificates. PolicyMaker [126] introduced a new policy language for specifying trusted actions and trust relationships, and a prototype for trust management system. In contrast to X.509 and PGP which map the certificate to the name and map the name to the actions allowed to perform, PolicyMaker integrates the policy specification with the direct binding of the public key to the predicates describing the authorized actions. Trust information is stored as assertions, which include the source of the assertions, the set of the public keys asserted, and the associated conditions which are security policies and credentials defined in terms of filter. The PolicyMaker acts as a query engine which receives policy statement, a set of credentials, and an action string from the application. Then, it determines whether the requested action string can satisfy the filter.

Keynote [129] is designed to authorize a principle to perform action(s) on behalf of the authorizer. It uses policy statements to delegate permissions, namely delegation credentials, which allow declarations of the delegation and the conditions of use.
Keynote policy is an assertion governing the delegation. It consists of the licensee, the authorizer, and an action attribute set which is a predicate that verifies the local variables. Keynote is a well-known language for trust management and is believed to be useful as a sub-language for specifying constraints on delegation of authority [127].

Role-based Trust Management Framework (RT) [130] provides RT family policy languages, semantics, deduction engine, and other pragmatic features to support distributed management of trust relations to determine the degree of trust associated with a peer, a service, or a coalition. Using logic-programming based approach, RT aims to provide access control and authorization based on policy statements made by multiple principals in large-scale decentralized systems. For example, the TM engine can induce the set of permissions granted to a particular requester. A main feature of RT framework is its distributed chain discovery technique [131] which allows the trust establishment process to start with an incomplete set of credentials. In RT, principals are identified by means of identity certificates. RT assigns role(s) to each credential and utilizes credential graphs for credential chain discovery. In the most basic form, $RT_0$ encodes policy statements as role definition credentials, which compose of a set of role definitions. Each role is unparameterized. $RT_1$ supports credentials with parameterized role. Each parameter can be a constant, a variable, or a value set of a defined type. $RT^T$ enables separation of duty using manifold roles. Each of which defines a set of principal sets using role expressions. Lastly, $RT^D$ includes delegation of role activations through delegation credentials.

Automated trust negotiation (ATN) enables online trust establishment between entities through a sequence of bilateral credential disclosure, to control access to sensitive resources (or credentials). It was first introduced to manage the exchange of credentials between strangers in property-based authentication and authorization [132]. Each participant has credential access policy (CAP) for each of its credentials that must be satisfied before the credential is disclosed. With Eager strategy, the participants immediately give all of their credentials whose CAPs are satisfied without the request from the opponent. Thus, this strategy unnecessarily reveals some credentials. In contrast, parsimonious strategy provides local minimality of disclosures. Credentials are released only by credential requests. However, the credential requests can reveal sensitive information about its own credentials and properties. Both strategies do not protect the CAPs.

The protection of credential access control policies was first addressed in TrustBuilder [133]. It uses policy graph instead of a single policy to represent resource’s access control to protect sensitive policies. The graph depicts the policies to be satisfied before revealing the sensitive policies. However, policy graph treats the policies
differently from resources. It also couples policy disclosure with policy satisfaction, which makes it hard to express certain access control requirements [134]. In addition, TrustBuilder addresses the issue of interoperability between different trust negotiation strategies [135, 128]. The work characterizes a broad class of strategies and designs a strategy-independent, language-independent trust protocol that ensure the interoperability is among the strategies in the class within the TrustBuilder negotiation architecture. An extension of TrustBuilder is to incorporate content-triggered trust negotiation to protect sensitive content disclosed by the clients [136]. The proposed access control model detects the transmission of sensitive information using content classification, maps the types of sensitive information to the access control policy using dynamic policy association (late binding), and determines whether the receiver is authorized to receive it using authorization component.

UniPro [134] extended the TrustBuilder’s policy graphs concept to create a unified scheme which protects the resources including the negotiation policies. It models the policies as first-class resources and distinguishes between policy disclosure and policy satisfaction, allowing the policy to be satisfied without disclosing it. Moreover, UniPro supports autonomous selection of negotiation strategies and strategy interoperability. The policy protection in UniPro comes with a trade-off between privacy and access. When some parts of the policy are hidden, the opponent may not try to disclose some more credentials to satisfy the policy.

TrustBuilder2 features CLOUSEAU policy compliance checker [137] which offers an efficient way to find all satisfying sets of credentials for a given policy by viewing the problem as a pattern-matching problem. A policy is specified as patterns that place constraints on the credentials or other evidence it needs. These credentials are viewed as a set of objects. The algorithm aims to match the set of objects to at least one of these patterns. TrustBuilder2 was also extended to bring trust negotiation to web services in a way that is compliant to the existing WS standards [138].

Trust-Serv [139, 140] also protects web services, through model-driven trust negotiation. Instead of expressing trust negotiation policies with a policy language, TrustServ uses state machine for policy representation to provide graphical model. Each state depicts the level of trust achieved so far. It is assigned with role abstraction, implying that the requester can activate the associated role when arriving at the state. The state transitions are labeled with conditions required to activate the role, which can be credential disclosure, provisions or obligations, and timeouts.

Trust-χ [141] is an XML-based trust negotiation framework for P2P environment.
The users create the XML-based $\chi$-TNL certificates [142] to describe their attributes and other uncertified declarations. They also specify $\chi$-TNL policies to protect their sensitive resources. Moreover, Trust-$\chi$ supports trust ticket to attest that the user has recently completed some negotiation and enable them to bypass some portions of the current negotiation.

2.7.3 Policy Reconciliation

Group communication systems such as collaborative applications and services handle multiple participants with different session requirements. In this environment, each participant expresses their requirements through a set of domain policies; each of which is a collection of statements defining the requirements on service provisioning and authorization. The policy reconciliation seeks for a policy instance, which is a session-defining specification that satisfies all of the domain policies from all of the participants [143].

Security Policy System (SPS) [144] supports two-party reconciliation. It was designed to resolve IPsec security associations between domains of communication. The reconciliation involves the intersection sets of policy values. Dynamic Cryptographic Context Management (DCCM) project [145] handles policy reconciliation among multiple parties. The security policies are negotiated between dynamically changing groups of participants. There are multiple rounds of negotiations between the participants to create a common policy. Ismene policy language [146] was created to allow the specification and reconciliation of group security requirements. Together with Ismene, the Antigone system was developed to enforce it [147]. With this, the work showed that general purpose provisioning policy reconciliation is intractable [143].

While the above policy reconciliation schemes required the participants to reveal their entire security policy, preference-maximizing privacy-preserving policy reconciliation [148] takes into account the privacy concerns of the two reconciliating parties.

2.7.4 Automated Policy Generation

Some body of literature looks into the area of automated security policy generation. A security assessment and policy templates generation tool called SAGEPOT [149] considers ISO 17799 as a reference standard. It interacts with the security auditor to assess the compliance of the system with the standard. The tool also generates the generic templates for the required security policies. Recently, an access-control policy generation technique for firewall policies was proposed [150]. From a given set of policies, the approach learns the policy parameters and builds synthetic policies which follow the
given grammar/syntax using a probabilistic learning approach. The generated policies can be used for policy configuration analysis and security devices testing. Chang et al. developed an approach for automatic generation of conflict-free IPSec policies [151]. They considered two main situations of policy conflicts and proposed algorithm sought for the correct configuration of policies while satisfying the security requirements.

2.8 Summary

This chapter provides essential background knowledge to promote the understanding of the mobile ecosystem and its inherent security issues. We gave an overview of the mobile ecosystem and introduce the major players in the environment. Considering its recent evolution, we deliberate over the impacts of the openness and the convergence on the security of the system and the classes of security concerns that each stakeholder has with the interest to protect themselves. Moreover, to realize the current security state of mobile phones and cellular networks, we describe several well-known vulnerabilities along with the research attempts and the real-world practices aiming at mitigating them.

As the Android mobile phone operating system is used as the medium in our research, we give an overview of the Android architecture, its application model, and its current security model. We put forward the fact that the Android security model also suffers from the two mobile phone security challenges, namely the lack of a clear definition of security, and the limited capability to incorporate the stakeholders’ security requirements into the model. As a result, it is insufficient for fulfilling the stakeholders’ expectation, which is an essential property of secure mobile phones.

Lastly, we present the fundamental of security requirements engineering, which are used in our research as an approach to develop the stakeholders’ security policies representing their security requirements. We specify four main factors that render the existing SRE methodologies inappropriate for our scenario, which leads to the need to develop an alternative approach to be applied to mobile phones. We end the chapter with the discussion on policy systems.
Securing Mobile Phone Platform Against Malware

The most fundamental element of mobile phones is the mobile phone platforms which include the operating systems and the built-in components. They perform all low-level operations and provide supporting functionality for all of the phone applications and services. Being the grounds for the data, resource, and transaction handling, the use of mobile phones cannot be secure if the platforms are prone to attacks. For this reason, the fidelity of the phone platforms is always the greatest concern. In this chapter, we explore security from the perspective of the phone platform. In particular, this part of our research investigates the threats from malicious applications, which is one of the most serious risks for the platforms. We demonstrate the benefits of taking the platform developers’ and the users’ security interests into building a malware prevention system.

Our study starts with a comprehensive discussion on the characteristics of malware and the attackers’ motivation for them which reflect the security concerns resulted from the successful malware exploits.

Thereafter, we identify the limitation of the current malware mitigation strategies which usually rely solely on external application certification process. We then propose a novel on-platform install-time application certification service called Kirin, which in turn secures the benign applications and ultimately the phone’s user. Achieving a practical solution requires overcoming multiple challenges. Certifying applications based on security configuration requires a clear specification of undesirable properties. While a piece of software is considered malware when it exhibits infiltrating and damaging behaviors, defining malware characteristics is still a non-trivial task. Particularly, we try
to accomplish this at install time before the software is executed.

One of our key hurdles is to identify the characteristics of the software that can potentially make the phone deviate from a secure phone. The accompanying part of the chapter describes how we address this challenge through the adoption of security requirements engineering. We present our methodology for developing the security policies that represent the stakeholders’ security requirements with respect to malware. The detailed security policies for mitigating malware are constructed from an analysis of applications, phone stakeholders, and systems interfaces. We also point out the limitations of existing security enforcement in Android which make practical policies difficult to define. We define a security language to encode these policies and formally define its semantics.

Lastly, to show that the incorporation of the stakeholders’ expectations on the applications is an effective avenue for promoting platform security, we evaluate our malware prevention system using the applications from the Android Market. We show that of the 311 evaluated applications spanning 16 categories, 10 were found to assert dangerous permissions. Of those 10, 5 were shown to be potentially malicious and therefore should be installed on a personal cell phone with extreme caution. The remaining 5 asserted rights that were dangerous, but were within the scope of reasonable functional needs (based on application descriptions). This analysis flagged about 1.6% applications at install time as potentially dangerous. Thus, even with conservative security policy, less than 1 in 50 applications needed any kind of involvement by phone users. We also include the vulnerabilities we discovered over the course of our study at the end of this chapter.

In the design and evaluation of Kirin, this part of the research makes the following contributions:

- We provide a methodology for retrofitting security requirements in Android. As a secondary consequence of following our methodology, we identified multiple vulnerabilities in Android, including flaws affecting core functionality such as SMS and voice.

- We provide a practical method of performing lightweight certification of applications at install time. This benefits the Android community, as the Android Market currently does not perform rigorous certification.

- We provide practical platform policies to mitigate malware. These policies are constructed purely from security configuration available in application package manifests.
The author’s major contributions to this work include the platform security policy development and the implementation of Kirin. The architectural design of Kirin and Kirin security language are mainly contributed by William Enck [152].

3.1 Understanding Malware Situation

As the mobile phone application market places are rapidly expanding, the mobile phone users feel more comfortable downloading games and applications. For instance, Apple’s App Store came to celebrate 1.5 billion downloads with over 65,000 applications after only one year from its birth [153]. The number of available applications rapidly rose to more than 130,000 applications at the beginning of 2010. It was also recorded that the users of Android phone, iPhone, and iPod Touch download nine, ten, and eighteen applications a month respectively on average [154]. Unfortunately, with the increased amount of applications downloading, the users experience a higher chance of installing malicious applications.

Malicious applications running on the phone can compromise other applications, the platform integrity, the phone’s data, and the network. As described in the previous chapters, the impacts of malware can range from minor irritation, stealing of confidential information, to complete system breakdown. Many expect mobile phone malware to begin following PC-based malware trends of fulfilling financial motivations [155]. Recently, it is evidenced that malicious applications can form a mobile phone botnet in a similar way to the botnet of computers. These zombie mobile phones respond to the bot master’s commands to send spams [156]. They can also potentially infiltrate computer network [157], or even bring down the cellular network [9].

3.1.1 Categories of Malware

To understand malware, mobile phone literature has studied and categorized mobile phone malware based on different criteria. Guo et al. [24] consider categories of resulting network attacks. Cheng et al. [69] derive models based on infection vector (e.g., Bluetooth vs. MMS). However, the most appropriate classification of malware for designing the platform security policies against them is the taxonomy based on an attacker’s motivations [158] because these incentives depict the outcomes that the security policies should address. The following is the brief list of motivations that we believe to seed future malware.

- *Proof-of-Concept:* This class of malware often emerges as new infection vectors
are explored by malware writers. While they are usually not created to cause any devastating effect, they frequently lead to unintended consequences. For instance, Cabir demonstrated Bluetooth-based distribution but inadvertently drained device batteries. In particular, as newly launched phone platforms become more popular, we often witness proof-of-concept malware as the initial threats for these platforms. As an example, following the release of iPhone in June 2007, the first iPhone Trojan, which appeared to be only a prank, was found during the first week of the year 2008 [159].

- **Destructive:** As illustrated in Section 2.4.2, malware such as Skulls and CardLock were designed with destructive motivations. While we believe malware with monetary incentives will overtake destructive malware, it will continue for the time being. Future malware may infect more than just the integrity of the phone since technology convergence would accelerate the propagation of any destructive impacts. For example, current phone operating systems and applications heavily depend on cloud computing for storage and reliable backup. If malware, for example, deletes entries from the phone’s address book, the data loss will propagate on the next cloud synchronization and subsequently affect all of the user’s computing devices.

- **Premeditated Spyware:** Spyware such as FlexiSpy [www.flexispy.com](http://www.flexispy.com) is marketed as a tool to “catch cheating spouses” and is available for Symbian, Windows Mobile, Black Berry, and iPhone. It provides location tracking, and remote listening. While malware variants exist, the software itself exhibits malware-like behavior and will likely be used for industrial espionage, amongst other purposes. Such malware may be downloaded and installed directly by the adversary, e.g., when the user leaves the phone on a table.

- **Direct Payoff:** Some malware directly compensates the malware’s authors. For example, masquerading as utility programs, Viver sends messages to premium SMS numbers [160]. We will undoubtedly see similar malware appearing more frequently due to economic incentive. Such attacks impact both the end-user and the provider. Customers will contest the additional fees, leaving the provider with the expense. Any mechanism providing direct payment to a third party is a potential attack vector. For example, Apple’s iPhone OS 3.0 has in-application content sales [161].

- **Information scavengers:** Web-based malware currently scourgs PCs for valuable address books and login credentials (e.g., usernames, passwords, and cookies for
two-factor authentication for bank websites) [162]. Mobile phones are much more organized than their PC counterparts. Interesting user data is in predictable locations, making them better targets for such malware [158]. For example, most phone operating systems include an API allowing all applications to directly access the address book.

- **Ad-ware:** Today’s Internet revenue model is based upon advertisements. The mobile phone market is no different, with many developers receiving compensation through in-application advertisements. We expect malware to take advantage of notification mechanisms (e.g., the Notification Manager in Android); however, their classification as malware will be controversial. Ad-ware on mobile phones is potentially more invasive than PC counterparts, because the mobile phone variant will use geographic location and potentially Bluetooth communication [158].

- **Botnet:** A significant portion of current malware activity results in a PC’s membership into a botnet. Many anticipate the introduction of mobile phones into botnets, even coining the term mobot (mobile bot) [163]. In line with Traynor’s prediction in 2008 on the existence of a mobile botnet in 2009 [164], the first botnet-alike malware was found in July 2009 [157, 156]. The goal of mobile botnets will most likely be similar to those of existing botnets (e.g., providing means of DoS and spam distribution); however, the targets will change. Core telephony equipment is expected to be subject to DoS by phones, and mobot-originated SMS spam will remove the economic disincentive for spammers, making SMS spam much more frequent and widespread. Finally, the phone functionality will be used. For example, telemarketers could use automated dialers from mobots to distribute advertisements, creating “voice-spam” [165].

### 3.1.2 Malware Mitigations and Limitations

As discussed in Section 2.5, the most commonly adopted malware mitigation thus far has been to ensure that only approved applications can be installed. In this scheme, a trusted evaluator, namely a certificate authority, devotes a massive amount of resources towards source code inspection, in order to verify the applications before it is released. In practice, given the currently prevalent application distribution channels, such certificate authority is the online market place (e.g. Apple’s App Store) or its alliance (e.g. SymbianSigned for Symbian Horizon). While this technique can screen out both malware and typical software misuse, it does not give any assurance. For example, the Symbian’s
express signing process basically provides anti-virus scanning. Only random samples are submitted for human audit [66]. This coarse procedure may overlook some malicious applications as mentioned in Section 2.5.

Moreover, the verification process is mainly based on the certifier’s own discretion. The application markets are usually associated with the platform providers. As a result, the applications are identified as dangerous normally in accordance with general security practices and the requirements of their business partners. For instance, software desired by the end users may be restricted by the network providers (e.g., VoIP and “Bluetooth tethering” applications). Nevertheless, these certifying authorities do not have enough knowledge to satisfy the security requirements of the stakeholders which they do not have pre-existing relationship with including the phone users.

In some circumstances, the applications available in the online market places cannot fulfill the user’s requirements, leaving the user with no choice but obtaining the applications from other sources. For Android, the Android Market yet does not certify applications. In such cases, the users cannot avoid installing uncertified applications and would be at risk.

This part of the dissertation seeks to protect the phone platform from malware and other software misuse while satisfying two main design objectives. First, the application certification process must take the users’ security requirements into account because the users’ expectation on the proper behavior of the applications may be different depending on their security and privacy concerns. Therefore, their definition of malware may vary. The approach should also support future extension to include the security requirements from other stakeholders. Second, we aim to avoid burdensome certification processes for each application.

### 3.2 Certifying Application at Install Time

In response to our two design goals, our proposed Kirin service performs lightweight certification at install time using a set of predefined security rules which constitute the platform policy. These rules decide whether or not the security configuration bundled with an application is safe.

#### 3.2.1 Application Installation in Android

We use the Android operating system as the instrument to validate our approach and to go over the implementation challenges. An advantage of application installation in
Android is the inclusion of the application’s property and security information in its application package, which is visible to the platform without the need to install the application. The current work is implemented at the application level. However, its open source nature allows us to modify its software installer engine in the future. Our ultimate goal is to embed the policy enforcement mechanism into the platform’s middleware to achieve mandatory security.

The overview of Android and its security model is presented in Chapter 2. In this section, we concentrate on some particular aspects of Android that is necessary to understand Kirin.

An Android application comes in form of Android Package (.apk) file. Every application package includes a manifest file. The manifest specifies all components in an application, including their types and Intent filters indicating the type of Intents they handle. Android also allows applications to dynamically create Broadcast Receivers that do not appear in the manifest. However, these components cannot be used to automatically start an application, as the application must be running to explicitly register them.

The manifest also includes security information. For the most part, security decisions are statically defined by the applications’ package manifests. Security policy in the package manifest centers on the concept of permission label. Once installed, this security policy cannot be changed. As cursorily introduced in Section 2.2.1, the policy primarily consists of the followings:

- **Permission declaration**: The application can define new permission labels to be added to the system. Normally, these new labels are used to protect its own components. The permission label must already be known to the system in advance or from this declaration in order for the permission request or use, as defined below, to be effective.

- **Permission request**: The application can request for permissions by their permission labels. These permissions are granted based on Android’s predefined permission granting policy described in Section 2.2.3. The application can be installed only if all the permissions it requests for are granted.

- **Permission use**: The application can assign a permission label to protect each of its components. With the exception of Content Providers, which can be protected with two permission labels for read and write.

At application install time, the manifest is examined by the software installer. If all the dangerous permissions are granted by the user, the installation process can continue.
Normal permissions are always granted. On the other hand, the assignment of Signature and SignatureOrSystem permissions can silently fail. Similarly, if the requested permissions are undefined, the permission granting also fails without informing the user. In these cases, the installation proceeds without granting the permissions.

After the installation is completed, the security policy is added to the system. The set of permissions declared by the application is also added. However, Android recognizes permissions by their labels. If the permission’s label already exists in the system, the permission is silently ignored. This silent failure due to duplicated permission declaration can lead to vulnerability as will be expanded on in Section 3.4.

In our research, we assume that the Android platform and the system applications bundled with the platform such as Phone, Messaging, Maps, and Browser constitute the Android’s trusted computing base (TCB). We aim to ensure that the application to be installed will not be harmful to the user and the Android’s TCB.

### 3.2.2 Application Certification with Kirin

The overwhelming number of existing malware requires manual installation by the user. While Bluetooth has provided the most effective distribution mechanism [155], as bulk data plans become more popular, so will SMS and email-based social engineering. Recently, Yxe propagated via URLs sent in SMS messages [166]. While certification from online application markets helps control mass application distribution, it is not a complete solution. Few (if any) existing phone malware exploits code vulnerabilities, but rather relies on user confirmation to gain privileges at installation.

Based on Android’s permission model, permission assignment is controlled by only two ways: user confirmation and signatures by developer keys. Android also uses “signature” permissions to prevent third-party applications from inflicting harm to the phone’s trusted computing base.

The Open Handset Alliance proclaims the design principle, “all applications are created equal.” This philosophy promotes innovation and allows manufacturers to customize handsets. However, in production environments, all applications are not created equal. Malware is the simplest counterexample. Once a phone is deployed, its platform, namely the trusted computing base, should remain fixed and must be protected. While it is clear for some particularly critical functionality to be protected with “signature” permissions, this may not be the case for some others. There is a trade-off when deciding if permission should be “dangerous” or “signature.” Initial Android-based production phones such as the T-Mobile G1 are marketed towards both consumers and developers. Without its
applications, Android has no clear competitive advantage. Google frequently chose the “feature-conservative” as opposed to “security-conservative” route and assigned permissions as “dangerous.” Nevertheless, some of these permissions may be considered “too dangerous” for a production environment. For example, one permission allows an application to debug others. Other times it is combinations of permissions that result in undesirable scenarios as will be discussed further in Section 3.3.

Kirin supplements Android’s existing security framework by providing a method to customize security for production environments. In Android, every application has a corresponding security policy. Kirin conservatively certifies an application based on its policy configuration. Certification is based on the platform security policy which consists of rules representing templates of undesirable security properties. Alone, these properties do not necessarily indicate malicious potential; however specific combinations allow malfeasance. For example, an application that can start on boot, read geographic location information, and access the Internet is potentially a tracker installed as premeditated spyware introduced above. It is often difficult for users to translate between individual properties and real risks. Kirin provides a means of defining dangerous combinations and automating analysis at install time.

Figure 3.1 depicts the Kirin embedded application installer. The installer first extracts security configuration from the target package manifest. Next, the Kirin security service evaluates the configuration against a collection of rules defined in the platform security policy. If the configuration fails to pass all rules, the installer has two choices. The more secure choice is to reject the application. Alternatively, Kirin can be enhanced with a user interface to override analysis results. Clearly this option is less secure for users who install applications without understanding warnings. However, we see Kirin’s analysis results as valuable input for a rating system similar to a web browser plug-in called PrivacyBird [167] that helps the user understand the privacy risk associated with
a specific website by interpreting its P3P policy. Such an enhancement for Android’s installer provides a distinct advantage over the existing method of user approval. Currently, the user is shown a list of all requested potentially dangerous permissions. A Kirin based rating system allows the user to make a more informed decision. Such a rating system requires careful investigation to ensure usability. We focus specifically on identifying potential harmful configurations and leave the rating system for future work.

3.2.3 Implementation of Kirin

For flexibility, Kirin is designed as a security service running on the mobile phone. The existing software installer interfaces directly with the security service. This approach follows Android’s design principle of allowing applications to be replaced based on manufacturer and consumer interests. More specifically, a new installer can also use Kirin.

We implemented Kirin as an Android application. The primary functionality exists within a Service component that exports an RPC interface used by the application installer. This service reads the platform security policy from a configuration file. At install time, the installer passes the file path to the package archive (.apk file) to the RPC interface. Then, Kirin parses the package to extract the security configuration stored in the package manifest. The PackageManager and PackageParser APIs provide the necessary information. The configuration is then evaluated against the rules in the platform security policy. Finally, the passed/failed result is returned to the installer with the list of the violated rules. Note that Kirin service does not access any critical resources of the platform hence does not require any permissions.

3.3 Developing Platform Security Policy

The malware threats and the Android architecture introduced in the previous sections serve as the background for developing platform security policy for detecting potentially dangerous application configurations. To identify the malware characteristics that impact the security of the Androids TCB, we need to define the term “secure phone” with respect to the malware problem. More specifically, we project the security requirements on the behaviors of safe applications. This is when security requirements engineering comes into play.
3.3.1 Security Requirements Engineering for Mobile Phones

Recall from Section 2.6 that security requirements in general are constraints or conditions on the system functions. They aim to protect the assets against potential threats which are the attacks that jeopardize the systems. In the context of mobile phones, the assets include not only the resources on the phones but also any resources that are deemed important to the users, the platform developers, the application developers, and the external providers, ranging from sensitive information, profits, reputation, to privacy.

Our security requirements engineering approach aims to develop security policies which represent the stakeholders’ security requirements. Primarily, each security policy statement corresponds to a security requirement which consists of constraints on actors and operations. In our methodology, prior to actually creating each security policy statement, we need to understand (i.) the asset we aim to protect, (ii.) the function(s) on that particular asset and its operating environment, and (iii.) the potential threats on the asset defined in relation to the specific system, the operating environment, and our domain of interest. However, the steps to which the three pieces of information are obtained are different depending on the available information that the stakeholders may have in hand. In response, our security policy development methodology consists of three prongs serving three domains of interest, namely the phone platform, the phone applications, and the content, which involves different sets of stakeholders. After the threats are identified, we develop the constraint(s) on the actor(s) and the operation(s) that prevent the realization of such threats. These are our security policies. Again, because the stakeholders’ interests are diverse across domains of interest, the set of actors and operations as well as the constraints may also be different.

In this chapter, we demonstrate our first prong, which is devoted to platform security policy. The prongs for application and content security policy are detailed in Chapter 4 and 5.

3.3.2 Platform Security Policy Development Methodology

The first prong of our methodology aims to develop security policies for protecting the phone platforms by preventing the installation of such malicious applications. We seek to screen out the applications that are capable of performing dangerous operations. The capability of Android applications depends on the set of permissions they request for and the set of Intents they potentially receive. From SRE aspect, the high-level threats come from the applications with offensive capability. The applications to be installed
Figure 3.2. Steps for Developing Phone Platform Security Policy.

... the actors which can perform any operations within their capabilities. The policy statements in the platform security policy are *Kirin security rules* which constrain the operations by placing the conditions on the capabilities that the applications can have.

Creating the security policies requires the information about the asset(s) in concern, the functions on the asset(s), and the threats. However, the phone’s usage model and functional requirements are not well defined. The properties of a secure mobile phone remain ambiguous. As a result, it is more convenient to start by determining the assets. Furthermore, as we apply the platform security policy at installation time, our construction of the policy must be based purely on the configuration information available in application package manifests. Figure 3.2 depicts our five-step procedure. It provides a guideline for the stakeholders, which in this case include the users and the platform developers, to develop the platform security policies.

**Step 1: Identify Assets**

We extract assets from several features of the Android platform. Google has already indirectly identified many assets in the form of permission labels protecting resources. We also consider the system’s Intent messages platform assets as many applications rely on them to operate.

We create an *asset table* initially consisted of asset, permission, and Intent action string columns. To enumerate the assets identified by permissions, we iterate through the list of Android’s predefined permissions. This list maps the permissions to the assets being protected. These assets mainly include the system APIs and the system application...
components. The assets and their protecting permissions are added to our table. For the system Intent messages, we go through the list of system’s predefined Intents and add them to our table along with their action strings. If they are protected by permissions, the permissions are also added to the table accordingly.

The APIs and Intent messages are considered tangible assets because they are system components. They also reflect other intangible assets. For example, “audio recorder” asset actually depicts that the user’s microphone input (e.g. voice conversation) is an asset. “GPS location API” asset reflects that the user’s location is an asset. These intangible assets are also added to the table along with the permissions and the action strings associated with the tangible assets from which they are derived. The output from Step 1 is the asset table.

**Step 2: Identify Functions on Assets**

Next, we carefully study each asset to manipulate all possible operations on it and specify the corresponding functional descriptions. A column for these descriptions is added to the asset table. This step is vital to our design, because not only the assets but also their functional descriptions are necessary to investigate realistic threats. They indicate how the assets interact with the rest of the phone and third-party applications, which in turn depict how the applications can maliciously impact the phone platform through the assets. As we work with an existing system, these existing functions reflect the functional requirements which describe how the phones are designed to operate.

In our investigation, we study Android’s documentations to understand how the system APIs and Intents are used or accessed. To aid the process, we use problem diagrams to explain the operations from both the phone and the application perspectives. We put the description of every function that involves the asset into its functional description column.

**Step 3: Determine Asset Security Goals and Threats**

For each asset in the asset table, we enumerate the attacker goals for compromising the asset. We use the malware motivations outlined in Section 3.1.1 to drive the attacker goals. They are considered against the high-level security goals which are confidentiality, integrity, and availability. Those violate the security goals are considered our threats.

Note that defining threat descriptions sometimes requires a level of creativity. It involves a red teaming exercise to envision what attack can be mounted to the assets. To aid this process, we use attack tree [110, 109]. The attacker goal of compromising the
asset is put on the top of the attack tree. Then, the goal is gradually broken down to the subgoals which must be achieved to accomplish the goal above them. This attack tree decomposes the threat to finally reveal the functions that the applications must perform to achieve the goal. The sets of required functions constitute our threat descriptions which will be added to the asset table.

**Step 4: Develop Asset’s Security Policy**

Security policy ensures that the conditions required by the threats cannot be met. Therefore, for each threat, we consider each of its required functions at a time. The function is compared against the functional descriptions column of the entire asset table to seek for the assets that must be accessed to complete such function. Then, from the table, the required permissions to access such assets and the Intent action strings that the malicious application needs to receive are considered. The rules included in the security policy prevent the application to be installed from having all of the required permissions and/or receiving all of the required Intent action strings.

**Step 5: Determine Security Mechanism Limitations**

Our final step caters to the practical limitations of our intended enforcement mechanism which involves static analysis at install time. First, we cannot ensure runtime support beyond what Android already provides. Second, we are limited to the information available in an application package manifest. For both reasons, we must refine our list of security rules. Some rules may simply not be enforceable. For instance, we cannot ensure only a fixed number of SMS messages are sent during some time period [168], because Android does not support history-based policies. Lastly, the security rules must also be mapped to the security configuration available in the package manifest. This usually consists of identifying the permission labels used to protect functionality. Finally, as shown in Figure 3.2, the iteration between Steps 4 and 5 is required to adjust the policy’s rules to work within our limitations.

**3.3.3 Sample Platform Security Policies**

By following our five-step procedure, we develop some sample security rules. Recall that each rule corresponds to a policy statement to be included in the platform security policy. As an example, in Step 1, Android defines the `RECORD_AUDIO` permission to protect its audio recorder. This API asset reflects another intangible asset, namely the microphone
In Step 2, we consider the possible functions. On the phone side, when the user receives an incoming call, the system broadcasts an Intent to the PHONE_STATE action string. On outgoing call, this Intent as well as the Intent to the NEW_OUTGOING_CALL action string are broadcasted. Figure 3.3 shows the problem diagram for the functions presented in the outgoing call scenario of our example. The dotted circles are functional requirements. The requirement (i.) and (ii.) involve the outgoing call functions. On the application’s side, the application can use MediaRecorder API to record the audio (requirement (iii.) in Figure 3.3). The application can detect the initiation of the calls by two means. First, it registers to be notified with changes in the phone’s call state from both incoming and outgoing calls via the PHONE_STATE Intent or PhoneStateListener. This option requires READ_PHONE_STATE permission. Second, the application registers to receive the NEW_OUTGOING_CALL Intent if it has PROCESS_OUTGOING_CALLS permission.

We develop the threat tree to model the threats in Step 3 as shown in Figure 3.4. The attacker’s goal is to eavesdrop on the microphone input during the call conversation which abuses its confidentiality security goal. Consider the level 2 of the tree, this threat requires the application to detect the call activity, record the microphone input, and send the record to the adversary over the Internet (i.e., premeditated spyware). The arc connecting the three child nodes depicts a conjunctive refinement. It means that all the sub-goals must be achieved. The corresponding threat description becomes, “spyware can breach the user’s privacy by detecting the phone call activity, recording the conversation,
and sending it to the adversary via the Internet.”

In response to the three sub-goals in the Step 3, we observe during the Step 4 that the application requires (i.) notification of an incoming or outgoing call, (ii.) the ability to record audio, and (iii.) access to the Internet. Therefore, our security policy statement becomes, “an application must not be able to receive phone state, record audio, and access the Internet.”

The last step considers Android’s specific information and limitation for enforcing the policy. Continuing down the threat tree, the level 3 is derived using the Android specific functional requirements, as shown in Step 2, to present the threats specific to Android. There are three alternatives to achieve the attacker goal, namely through the leave nodes \{3.A, 2.B, 2.C\}, \{3.B, 2.B, 2.C\}, or \{3.C, 2.B, 2.C\}. To compromise the threats, the application must have the capabilities to perform the actions at the leave nodes. The actions in node 3.A and 3.B require \textit{READ_PHONE_STATE} permission. The actions in node 3.C, 2.B, and 2.C require permission \textit{PROCESS_OUTGOING_CALLS}, \textit{RECORD_AUDIO}, and \textit{INTERNET} accordingly. Therefore, the resulting security rules from our running example are: a) “an application must not have the \textit{READ_PHONE_STATE}, \textit{RECORD_AUDIO}, and \textit{INTERNET} permissions.” and the nearly identical b) “an application must not have the \textit{PROCESS_OUTGOING_CALLS}, \textit{RECORD_AUDIO}, and \textit{INTERNET} permissions.” The precise policy statements are rule 2 and 3 in Table 3.1.

Table 3.1 also shows some other sample security rules. We loosely categorize them by their complexity.

**Single Permission Security Rules**

A number of Android’s “dangerous” permissions may be “too dangerous” for some pro-
An application must not have the **SET_DEBUG_APP** permission label.

An application must not have **PHONE_STATE**, **RECORD_AUDIO**, and **INTERNET** permission labels.

An application must not have **PROCESS_OUTGOING_CALL**, **RECORD_AUDIO**, and **INTERNET** permission labels.

An application must not have **ACCESS_FINE_LOCATION**, **INTERNET**, and **RECEIVE_BOOT_COMPLETE** permission labels.

An application must not have **ACCESS_COARSE_LOCATION**, **INTERNET**, and **RECEIVE_BOOT_COMPLETE** permission labels.

An application must not have **RECEIVE_SMS** and **WRITE_SMS** permission labels.

An application must not have **SEND_SMS** and **WRITE_SMS** permission labels.

An application must not have **INSTALL_SHORTCUT** and **UNINSTALL_SHORTCUT** permission labels.

An application must not have the **SET_PREFERRED_APPLICATION** permission label and receive Intents for the **CALL** action string.

Table 3.1. Sample security rules for malware mitigation

We discovered several such permission labels. For instance, the **SET_DEBUG_APP** permission “...allows an application to turn on debugging for another application.” (according to the documentation). A malware author can simply download Android’s source code and build an SDK that includes the API. The malware then, for instance, can disable anti-virus software. Rule 1 ensures third party applications do not have the **SET_DEBUG_APP** permission. Similar rules can be applied to other permission labels protecting hidden APIs such as Bluetooth APIs which are not yet mature enough for general use [169].

Multiple Permission Security Rules

Some combinations of permissions enable the applications to perform dangerous operations. Therefore, we must define rules with respect to multiple permissions. To do this, we consider the minimal set of functionality required to compromise each particular threat. Rule 2 and 3 protect against the voice call eavesdropper previously shown. Similarly, Rule 4 and 5 protect against a location tracker. In this case, the malware starts executing on boot. In these security rules, we assume the malware starts on boot by
defining a Broadcast Receiver to receive the `BOOT_COMPLETE` action string. Note that the
`RECEIVE_BOOT_COMPLETE` permission label protecting this broadcast is a “normal” per-
mission (and hence is always granted). However, the permission label provides valuable
insight into the potential functions of an application.

Rule 6 and 7 consider malware’s interaction with SMS. Rule 6 protects against mal-
ware hiding or otherwise tampering with incoming SMS messages. For example, SMS
can be used as a control channel for the malware. However, the malware author does
not want to alert the user, therefore immediately after an SMS is received from a specific
sender, the SMS Content Provider is modified. In practice, we found that our sample
malware could not remove the SMS notification from the phone’s status bar. However,
we were able to modify the contents of the SMS message in the Content Provider. While
we could not hide the control message completely, we were able to change the message
to appear as spam. Alternatively, a similar attack could ensure the user never receives
SMS messages from a specific sender, for instance PayPal or a financial institution. Such
services often provide out-of-band transaction confirmations. Blocking an SMS message
from this sender could hide other activity performed by the malware. While this attack
is also limited by notifications in the status bar, again, the message contents can be
transformed as spam. Rule 7 mitigates mobile bots sending SMS spam. This rule en-
sures the malware cannot remove traces of its activity. While rule 7 does not prevent the
SMS spam messages from being sent, it increases the probability that the user becomes
aware of the activity.

Finally, Rule 8 makes use of the duality of some permission labels. Android defines
separate permissions for installing and un installing shortcuts on the phone’s home screen.
This rule ensures that a third-party application cannot have both. If an application has
both, it can redirect the shortcuts for frequently used applications to a malicious one.
For instance, the shortcut for the web browser could be redirected to an identically
appearing application that harvests passwords.

Permission and Interface Security Policies

Permissions alone are not always enough to characterize malware behavior. Rule 9 pro-
vides an example of a rule considering both a permission and an action string. This
specific rule prevents malware from replacing the default voice call dialer application
without the user’s knowledge. Normally, if Android detects two or more applications
contain Activities to handle an Intent message, the user is prompted which application
to use. This interface also allows the user to set the current selection as default. How-
ever, if an application has the `SET_PREFERRED_APPLICATION` permission, it can set the default without the user’s knowledge. Google marks this permission as “dangerous”; however, users may not fully understand the security implications of granting it. Rule 9 combines this permission with the existence of an Intent filter receiving the `CALL` action string. Hence, we can allow a third-party application to obtain the permission as long as it does not also handle voice calls. Similar rules can be constructed for other action strings handled by the platform and the system applications which constitute our trusted computing base.

### 3.3.4 Kirin Security Language

We now describe the Kirin Security Language (KSL) to encode security rules for the Kirin service. Kirin uses an application’s package manifest as input. The rules identified in Section 3.3.3 only require knowledge of the permission labels requested by an application and the action strings used in Intent Filters. This section defines the KSL syntax and formally defines its semantics.

**KSL Syntax**

Figure 3.5 defines the Kirin Security Language in BNF notation. A KSL rule-set consists of a list of rules. A rule indicates combinations of permission labels and action strings that should not be used by third-party applications. Each rule begins with the keyword “`restrict`”. The remainder of the rule is the conjunction of sets of permissions and action strings received. Each set is denoted as either “`permission`” or “`receive`”, respectively.

**KSL Semantics**

We now define a simple logic to represent a set of rules written in KSL. Let $\mathcal{R}$ be set
of all rules expressible in KSL. Let $P$ be the set of possible permission labels and $A$ be the set of possible action strings used by Activities, Services, and Broadcast Receivers to receive Intents. Then, each rule $r_i \in R$ is a tuple $(2^P, 2^A)$.\footnote{We use the standard notation $2^X$ represent the power set of a set $X$, which is the set of all subsets including $\emptyset$.} We use the notation $r_i = (P_i, A_i)$ to refer to a specific subset of permission labels and action strings for rule $r_i$, where $P_i \in 2^P$ and $A_i \in 2^A$.

Let $R \subseteq R$ correspond to a set of KSL rules. We construct $R$ from the KSL rules as follows. For each $\langle \text{rule} \rangle_i$, let $P_i$ be the union of all sets of "permission" restrictions, and let $A_i$ be the union of all sets of "receive" restrictions. Then, create $r_i = (P_i, A_i)$ and place it in $R$. The set $R$ directly corresponds to the set of KSL rules and can be formed in time linear to the size of the KSL rule set (proof by inspection).

Next we define a configuration based on package manifest contents. Let $C$ be the set of all possible configurations extracted from a package manifest. We need only capture the set of permission labels used by the application and the action strings used by its Activities, Services, and Broadcast Receivers. Note that the package manifest does not specify action strings used by dynamic Broadcast Receivers; however, we use this fact to our advantage (as discussed in Section 3.4). We define configuration $c \in C$ as a tuple $(2^P, 2^A)$. We use the notation $c_t = (P_t, A_t)$ to refer to a specific subset of permission labels and action strings used by a target application $t$, where $P_t \in 2^P$ and $A_t \in 2^A$.

We now define the semantics of a set of KSL rules. Let $fail : C \times R \rightarrow \{\text{true, false}\}$ be a function to test if an application configuration fails a KSL rule. Let $c_t$ be the configuration for target application $t$ and $r_i$ be a rule. Then, we define $fail(c_t, r_i)$ as:

$$(P_t, A_t) = c_t, (P_i, A_i) = r_i, P_i \subseteq P_t \land A_i \subseteq A_t$$

Clearly, $fail(\cdot)$ operates in time linear to the input, as a hash table can provide constant time set membership checks.

Let $F_R : C \rightarrow R$ be a function returning the set of all rules in $R \in 2^R$ for which an application configuration fails: $F_R(c_t) = \{r_i | r_i \in R, fail(c_t, r_i)\}$

Then, we say the configuration $c_t$ passes a given KSL rule-set $R$ if $F_R(c_t) = \emptyset$. Note that $F_R(c_t)$ operates in time linear to the size of $c_t$ and $R$. Finally, the set $F_R(c_t)$ can be returned to the application installer to indicate which rules failed. This information
facilitates the optional user override extension described in Section 3.2.

### 3.4 Evaluating Platform Protection Against Malware

Practical security rules must both mitigate malware and allow legitimate applications to be installed. Section 3.3.3 argued that our sample security rules can detect specific types of malware. However, our evaluation involves static analysis which cannot disclose the runtime behavior of the application. As a result, we consider the worst-case scenario which assumes that the application would perform any actions as allowed by its capabilities. The applications that fail to comply with the platform policies will not be installed. As Kirin’s certification technique conservatively detects dangerous functionality, it may reject legitimate applications. In this section, we evaluate our sample security rules against real applications from the Android Market. While the Android Market does not perform rigorous certification, we initially assume it does not contain malware. Any application not passing a security rule requires further investigation. Overall, we found very few applications where this was the case. On one occasion, we found a rule could be refined to further reduce this number.

Our sample set consisted of a snapshot of a subset of popular applications available in the Android Market in late January 2009. We downloaded the top 20 applications from each of the 16 categories, producing a total of 311 applications (one category only had 11 applications). We used Kirin to extract the appropriate information from each package manifest and ran the $F_R(\cdot)$ algorithm described in Section 3.3.4.

#### 3.4.1 Empirical Results

Our analysis tested all 311 applications against the security rules listed in Table 3.1. Of the 311 applications, only 12 failed to pass all 9 security rules. Of these, 3 applications failed Rule 2 and 9 applications failed Rules 4 and 5. These failure sets were disjoint, and no applications failed the other six rules.

Table 3.2 lists the applications that fail Rule 2. Recall that Rule 2 defends against

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walki Talkie Push to Talk</td>
<td>Walkie-Talkie style voice communication.</td>
</tr>
<tr>
<td>Shazam</td>
<td>Utility to identify music tracks.</td>
</tr>
<tr>
<td>Inauguration Report</td>
<td>Collaborative journalism application.</td>
</tr>
</tbody>
</table>

Table 3.2. Applications failing Rule 2
a malicious eavesdropper by failing any application that can read phone state, record audio, and access the Internet. However, none of the applications listed in Table 3.2 exhibit eavesdropper-like characteristics. Considering the purpose of each application, it is clear why they require the ability to record audio and access the Internet. We initially speculated that the applications stop recording upon an incoming call. However, this was not the case. We disproved our speculation for Shazam and Inauguration Report and were unable to determine a solid reason for the permission label’s existence, as no source code was available.

After realizing that simultaneous access to phone state and audio recording is in fact beneficial (i.e., to stop recording on incoming call), we decided to refine Rule 2. Our goal is to protect against an eavesdropper that automatically records a voice call on either incoming or outgoing call. Recall that there are two ways to obtain the phone state: 1) register a Broadcast Receiver for the PHONE_STATE action string, and 2) register a PhoneStateListener with the system. If a static Broadcast Receiver is used for the former case, the application is automatically started on incoming and outgoing call. The latter case requires the application to be already started, e.g., by the user, or on boot. We need only consider cases where it is started automatically. Using this information, we split Rule 2 into two new security rules. Each appends an additional condition. The first appends a restriction on receiving the PHONE_STATE action string. Note that since Kirin only uses Broadcast Receivers defined in the package manifest, we will not detect dynamic Broadcast Receivers that cannot be used to automatically start the application. The second rule appends the boot complete permission label used for Rule 4. Rerunning the applications against our new set of security rules, we found that only the Walkie Talkie application failed our rules, thus reducing the number of failed applications to 10.

Table 3.3 lists the applications that fail Rules 4 and 5. Recall that these security rules detect applications that start on boot and access location information and the Internet. The goal of these rules is to prevent location tracking software. Of the nine applications listed in Table 3.3, the first five provide functionality that directly contrast with the rule’s goal. In fact, Kirin correctly identified both AccuTracking and GPS Tracker as dangerous. Both Loopt and Twidroid are popular social networking applications; however, they do in fact provide potentially dangerous functionality, as they can be configured to automatically start on boot without the user’s knowledge. Finally, Pintail is designed to report the phone’s location in response to an SMS message with the correct password. While this may be initiated by the user, it may also be used by an adversary to track the user. Again, Kirin correctly identified potentially dangerous functionality.
Application Description

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccuTracking</td>
<td>Client for real-time GPS tracking service (AccuTracking).</td>
</tr>
<tr>
<td>GPS Tracker</td>
<td>Client for real-time GPS tracking service (InstaMapper).</td>
</tr>
<tr>
<td>Loopt</td>
<td>Geosocial networking application that shares location with friends.</td>
</tr>
<tr>
<td>Twidroid</td>
<td>Twitter client that optionally allows automatic location tweets.</td>
</tr>
<tr>
<td>Pintail</td>
<td>Reports the phone location in response to SMS message.</td>
</tr>
<tr>
<td>WeatherBug</td>
<td>Weather application with automatic weather alerts.</td>
</tr>
<tr>
<td>Homes</td>
<td>Classifieds application to aid in buying or renting houses.</td>
</tr>
<tr>
<td>T-Mobile Hotspot</td>
<td>Utility to discover nearby nearby T-Mobile WiFi hotspots.</td>
</tr>
<tr>
<td>Power Manager</td>
<td>Utility to automatically manage radios and screen brightness.</td>
</tr>
</tbody>
</table>

* Did not fail Rule 5

Table 3.3. Applications failing Rule 4 and 5

The remaining four applications in Table 3.3 result from the limitations in Kirin’s input. That is, Kirin cannot inspect how an application uses information. In the previous cases, the location information was used to track the user. However, for these applications, the location information is used to supplement Internet data retrieval. Both WeatherBug and Homes use the phone’s location to filter information from a website. Additionally, there is little correlation between location and the ability to start on boot. On the other hand, the T-Mobile Hotspot WiFi finder provides useful functionality by starting on boot and notifying the user when the phone is near such wireless networks. However, in all three of these cases, we do not believe access to “fine” location is required; location with respect to a cellular tower is enough to determine a city or even a city block. Removing this permission would allow the applications to pass Rule 4. Finally, we were unable to determine why Power Manager required location information. We initially thought it switched power profiles based on location, but did not find an option.

In summary, 12 of the 311 applications did not pass our initial security rules. We reduced this to 10 after revisiting our security requirements engineering process to better specify the rules. This is the nature of security requirements engineering, which an ongoing process of discovery. Of the remaining 10, Kirin correctly identified potentially dangerous functionality in 5 of the applications, which should be installed with extreme caution. The remaining five applications assert a dangerous configuration of permissions, but were used within reasonable functional needs based on application descriptions.
Therefore, Kirin’s conservative certification technique only requires user involvement for approximately 1.6% of applications (according to our sample set). From this, we observe that Kirin can be very effective at practically mitigating malware.

### 3.4.2 Mitigating Malware

We have shown that Kirin can practically mitigate certain types of malware. However, Kirin is not a complete solution for malware protection. We constructed practical security by considering different malicious motivations. Some motivations are more difficult to practically detect with Kirin. Malware of destructive or proof-of-concept origins may only require one permission label to carry out its goals. For example, malware might intend to remove all contacts from the phone’s address book. Kirin cannot simply deny all third-party applications the ability to write to the address book. Such a rule would fail for an application that merges Web-based address books (e.g., Facebook).

Kirin is more valuable in defending against complex attacks requiring multiple functionalities. We discussed a number of rules that defend against premeditated spyware. Rule 8 defends against shortcut replacement, which can be used by information scavengers to trick the user into using a malicious Web browser. Furthermore, Rule 6 can help hide financial transactions that might result from obtained usernames and passwords. Kirin can also help mitigate the effects of botnets. For example, Rule 7 does not let an application hide outbound SMS spam. This requirement can also be used to help a user become aware of SMS sent to premium numbers (i.e., direct payoff malware). However, Kirin could be more effective if Android’s permission labels distinguished between sending SMS messages to contacts in the address book versus arbitrary numbers.

Kirin’s usefulness to defend against ad-ware is unclear. Many applications are supported by advertisements. However, applications that continually pester the user are undesirable. Android does not define permissions to protect notification mechanisms (e.g., the status bar), but even with such permissions, there are many legitimate reasons for using notifications. Despite this, in best case, the user can identify the offending application and uninstall it.

### 3.4.3 Discovered Vulnerabilities

The process of retrofitting security requirements for Android platform with respect to malware had secondary effects. In addition to identifying rules for Kirin, the intensive study of the phone’s assets and Android’s functionality and security mechanism reveals a
number of configuration and implementation flaws. This section outlines the discovered vulnerabilities based on the following categories:

**Unprotected System Assets:** From Step 1 which identifies assets, we found that not all assets are protected by permissions. In particular, in early versions of our analysis discovered that the Intent message broadcasted by the system to the **SMS_RECEIVED** action string was not protected. Hence, any application can create a forged SMS that appears to have come from the cellular network. Upon notifying Google of the problem, the new permission **BROADCAST_SMS_RECEIVED** has been created and protects the system broadcast as of Android version 1.1. We also discovered an unprotected Activity component in the phone application that allows a malicious application to make phone calls without having the **CALL_PHONE** permission. This configuration flaw has also been fixed. However, we still found 37 unprotected critical Intents in the SDK 1.1 r.1, which is the latest release at the time of experiment.

**Unchecked Permissions:** Google’s documentation describes the protection that each permission provides. Nevertheless, our investigation revealed that four critical permissions are never checked and enforced, as shown in Table 3.4. Consequently, malicious applications can use the resources that these permissions aim to protected without requesting for the permissions, i.e. without the user’s knowledge.

**Silent Failure:** We found that duplicate declaration of permission leads to vulnerabilities. The Android platform identifies the permission by its label. Without informing the user, the system silently ignores the duplicated permission and uses the first declared permission, which may have weaker protection level. Moreover, if the user installs the service with duplicated Intent Filter, the service will be silently ignored. The first installed service with that particular Intent Filter will always be the only service that receive the Intent. Similar silent failure is applied to Content Providers with duplicated authority string.

All of these flaws show the value in defining security requirements. Kirin relies on Android to enforce security at runtime. Ensuring the security of a phone requires a complete solution, of which Kirin is only part.
<table>
<thead>
<tr>
<th>Permission</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAMERA</td>
<td>Required to be able to access the camera device.</td>
</tr>
<tr>
<td>MODIFY_AUDIO_SETTINGS</td>
<td>Allows an application to modify global audio settings.</td>
</tr>
<tr>
<td>RECORD_AUDIO</td>
<td>Allows an application to record audio.</td>
</tr>
<tr>
<td>READ_PHONE_STATE</td>
<td>Allows read only access to phone state. It can be bypassed by directly registering with PhoneStateListener instead of registering to receive PHONE_STATE Intents.</td>
</tr>
</tbody>
</table>

Table 3.4. Unchecked Permissions (Experiment on SDK 1.1 Release 1)

3.5 Summary

Mobile phones must be secure from their fundamental which includes the phone platforms. As evidenced from the real world attacks on mobile phones, one of the most hostile threats against the platforms is malware, especially these days when the users become more comfortable downloading applications and games for their phones.

This chapter investigated the malware problem, pointed out the limitation of the current application certification practices, proposed Kirin, a lightweight on-platform certification approach. Kirin identifies potentially dangerous applications at install time without the need for burdensome code inspection for each application. In particular, we demonstrate the incorporation of the stakeholders’ security requirements on the characteristics of the well-behaved applications into the application certification process. Through our SRE-based platform security policy development methodology, we explore appropriate sets of policy statements, namely Kirin security rules, which are enforced on the applications being installed by the Kirin service.

We have shown that Kirin can express meaningful policy statements for mitigating malware. Furthermore, we have shown that Kirin’s conservative certification technique rarely identifies reasonable use of asserted rights as potentially dangerous. This indicates that Kirin requires minimal user interaction in practice. Future work will continue our platform security policy development process to discover additional rules to defend against malware.
Securing Mobile Phone Applications in Open and Converging Model

Stepping to the upper layer of the mobile phone architecture, our next level of security concerns is projected from the perspective of mobile phone applications. The rising number of new applications that entered the market and of the applications downloaded by the users over the previous years [5, 154] reflects the phone users’ demand for extended functionality beyond that is provided by the platform and the embedded system applications. Since the security requirements of these applications are still being understood, the security infrastructure available for them in current smartphone operating systems is largely underdeveloped. In particular, the current move towards open and converging application model has allowed applications to interact and share data and functionality with an aim to create more innovative applications. However, this evolution has opened an opportunity for applications to compromise one another.

The existing mobile phone security models including Android’s are still insufficient for handling such evolving application model. For Android, its security model supports application policies which mainly control how the application can interact with the platform and other applications via the Android’s IPC. Nevertheless, the capability of the supported policy is still very limited, both in terms of expressibility and enforcement power as it is based solely on the straight permission model. This permission model is static and very coarse-grained. It is also too semantically limited for applications to define their own security policies in a meaningful way. More importantly, it mostly depends on the users and the platform, which have very limited knowledge about the operations of the applications, to make decision on permission granting and access control for the
applications. As a result, the application policies are rather system and user-centric.

This chapter investigates how the evolving application model impacts the way the applications are designed and operate, and the security ramifications. Accordingly, we then explore the application developers' security requirements for protecting their applications, their customers, and potentially their backend infrastructure. We demonstrate the advantage of incorporating these requirements into the application security model through more semantically rich application security policies. To realize our vision, we propose Secure Application INTeraction (Saint) framework. Saint is a modification of the Android platform to govern install-time permission assignment and runtime application interactions as dictated by the policy bundled with the application. This policy reflects the security interests of the application’s stakeholders. The accompanying section elaborates on the security requirements engineering-based application policy development and presents an in-depth description of the semantics of the application policy. Later, we provide the architecture and technical detail of Saint together with the areas for extension, optimization, and improvement. We conclude the chapter with the illustration of the usefulness of the Saint framework using real world application project and our performance evaluation results.

4.1 Evolving Application Model

Mobile phone applications support vast new markets in communication, entertainment, and commerce. Hardware and software for these applications are now widely available and often surprisingly inexpensive, such as those distributed through online application market places such as Apple’s App Store, Android Market, and BlackBerry App World.

The mobile phone application model has evolved as the mobile ecosystem has become more open and converging. Correspondingly, mobile phone applications have shifted from stand-alone designs to a collaborative (service) model. In this emerging environment, applications expose selected internal features to other applications, and use those provided by others. In the latter case, applications simply search and use appropriate providers of a service type at run-time, rather than bind itself to specific implementations during development. This style of application integration allows a rich culture of “use and extend” development that has led to an explosion of innovative applications and services. Android’s application model together with the Android application development community is possibly the best representation of this evolution, which promotes the Android’s principle “Breaking down application boundaries”. An application can fully expose its
data and functionality to be used by other applications. By combining information and functionality across multiple applications, this design encourages the creation of new and innovative applications. Apart from Android, such feature has also been included in other mobile phone operating systems such as Access Linux Platform (ALP) [170].

While taking down the barriers between applications gives a lot of benefits, security is inevitably a trade-off for such functionality. The design has broken down the protection shell around each application. Isolating applications not only guards against attacks from malicious applications but also contains vulnerabilities within unhealthy applications which would prevent the propagation of security problems such as the cross-application attacks introduced in Section 2.4.2. In contrast, through interaction, untrusted applications can interfere with other applications. Moreover, compromised applications can also contaminate healthy applications.

The security model of the Android system and that of many other phone operating systems is “system-centric”. Applications statically identify the permissions that govern the rights to their data and interfaces at installation time. However, the application developer has no ability thereafter to govern to whom those rights are given or how they are later exercised. In essence, permissions are asserted as often vague suggestions on what kinds of protections the application desires. The application cannot identify its trust relationship with other applications. It must take on faith that the operating system and user make good choices about which applications to give those permissions – which in many cases is impossible because they do not have sufficient context to do so.

Consider a hypothetical PayPal service built on Android. Applications such as browsers, email clients, software marketplaces, music players, etc. use the PayPal service to purchase goods. The PayPal service in this case is an application that asserts permissions that must be granted to the other applications that use its interfaces. What is a legitimate application? Only PayPal the application (really PayPal the corporation) is in a position to know the answer to that question. This is more than simply a question of who is making the request (which in many cases in Android is itself unknowable), but also where, when, how, and under what conditions the request is being made. Unfortunately, Android does not provide any means for answering those questions or enforcing a security policy based upon them. Simply put, the Android system protects the phone from malicious applications, but provides severely limited infrastructure for applications to protect themselves.

We observe three essential application policies not available to applications in the Android security framework:
1. **Permission assignment policy** - Applications have no ability to control to whom permissions are given to access their internal interfaces, e.g., white or blacklist applications that may invoke them.

2. **Interface exposure policy** - Android provides only rudimentary facilities for applications to control how their interfaces are used by other applications.

3. **Interface use policy** - Applications often have no means of selecting, at run-time, which application’s interfaces they use.

**4.2 Protecting Applications’ Interactions**

Because it is the application developers that have enough understanding about the goals and the functionality of the applications to identify the above three pieces of information, we need an approach to allow the application developers to assimilate them into the application policy and to enforce them during the execution.

This part of our research adopts security requirements engineering principle into developing a methodology to allow the application developers to create rich application policies which address these key application requirements. The Secure Application INTeraction (Saint) framework extends the existing Android security architecture to support these policies. In the augmented infrastructure, applications provide installation-time policies that regulate the assignment of permissions that protect their interfaces. At run-time, access of or communication between applications is subject to security policies asserted by the both that caller and callee applications. Saint application policies go far beyond the static permission checks currently available in Android by restricting access based on run-time state, e.g., location, time, phone or network configuration, etc.

**4.2.1 Motivating Example**

Consider a motivating example depicted in Figure 4.1. We present the PersonalShopper smartphone shopping application. PersonalShopper tracks the items a user wishes to buy and interacts with payment applications to purchase them. A user enters desired items through the phone’s user interface (potentially by clicking on items on a browser, media player, etc.), creating a vendor independent “shopping cart”. Users subsequently acquire items in one of two ways. The user can direct the application to “find” an item by clicking on it. In this case the application will search known online vendors or shopping search sites (e.g., Google Product Search) to find the desired item. Where multiple vendors
Figure 4.1. The PersonalShopper application finds desired items at the discretion of the user and interacts with vendors and payment applications to purchase them.

provide the same item, the user selects their vendor choice through a menu. The second means for finding a product is by geography—a user moving through, for example, a mall can be alerted to the presence of items available in a physical store by a location-based search application. In this case, she will be directed to the brick-and-mortar vendor to obtain the item.

Regardless of how the item is found, PersonalShopper’s second objective is to facilitate the purchase process itself. In this case, it works with our example checkout applications SecurePayer and TrustCheckout. PersonalShopper accesses checkout applications and acts as an intermediary between the buyer and the merchants to both improve the efficiency of shopping and to protect customer privacy. The application and the services they use will interact with password vaults to provide authenticating credentials. Following their completion, the transactions are recorded in a personal ledger application.

Consider a few (of many) security requirements this application suggests:

1. PersonalShopper should only use trusted payment services. In the example, it may trust SecurePayer and TrustCheckout, but does not trust other unknown payment providers (e.g., the M-Payer provider in Figure 4.1).

2. PersonalShopper may only want to restrict the use of the service to only trusted networks under safe conditions. For example, it may wish to disable searches while the phone is roaming or highly exposed areas (such as in public airports) or while battery is low.

3. PersonalShopper may require certain versions of service software be used. For example, the password vault application version 1.1 may contain a bug that leaks
password information. Thus, the application would require the password vault be version 1.2 or higher.

4. PersonalShopper may wish to ensure transaction information is not leaked by the phone’s ledger application. Thus, the application wishes to only use ledgers that don’t have access to the Internet.

5. Security requirements may be placed on PersonalShopper by the applications and services it uses. For example, in the interest of preserving location privacy, the location-based search application may only provide PersonalShopper location information only where PersonalShopper possesses the permissions to access location information itself, e.g., the phone’s GPS service.

None of these policies are supported by the current Android security system. While some of these may be partially emulated using combinations of complex application code, code signing, and permission structures, they are simply outside the scope of Android’s security policy. As a consequence (and core to our extensive experience building systems in Android), applications must cobble together custom security features on top of the rudimentary structures currently provided by the Android system. Where possible at all, this process is ad hoc, error prone, repetitive, and inexact. What is needed is for Android to provide applications a more semantically rich policy infrastructure.

4.2.2 Designing Saint Framework

Based on the three elements that are absent from the current security model for applications, namely permission assignment policy, interface exposure policy, and interface use policy, Saint is designed to provide two major functions. First, Saint supports the regulation of permission assignment at install time; and second, Saint supports the control of interface exposure and use during runtime. This section provides an overview of Saint.

4.2.2.1 Install-time Application Policy Support

To allow application developers to regulate how the permissions declared by their applications are granted to other applications, Saint’s install-time support regulates granting of application defined permissions. More specifically, an application declaring permission $P$ defines the conditions under which $P$ is granted to other applications at install-time. Conceptually, the application requesting the permission $P$ can be installed only if the policy for acquiring $P$ is satisfied. Saint represents a substantial departure from existing
Android permission assignment. The existing Android model allows/disallows a permission assignment based on application-independent rules, or where such rules provide insufficient guidance, user input. Conversely, Saint allows applications to exert control over the assignment of permissions it declares through explicit policy.

Depicted in Figure 4.2, install-time policies are enforced by the Saint installer based on decisions made by the AppPolicy provider, which maintains a database of all the install and run-time policies. Upon installing an application, the Saint-enhanced Android installer retrieves the requested permissions from the manifest file (step A). For each permission, it queries the AppPolicy provider (step B). The AppPolicy provider consults its policy database, and returns a decision based on matching rules (step C). If the policy conditions hold, the installation proceeds, otherwise it is aborted. Finally, on successful installation, the new application’s install-time and runtime polices are appended to the AppPolicy provider’s policy database.

4.2.2.2 Runtime Application Policy Support

Similarly, Saint enables application developers to gain better control over how their applications’ internal functionality is used by other applications as well as how they select the application to perform some functions for them. Saint’s runtime support regulates the interaction of software components within Android’s middleware framework. Any such interaction involves a caller application that sends the IPC and callee (B) application that receives that IPC. The IPC is allowed to continue only if all policies supplied by both the caller and callee are satisfied.

Depicted in Figure 4.2, the Saint policy infrastructure works as follows. The caller application A initiates the IPC through the middleware framework (step 1). The IPC is
intercepted by the Saint policy enforcement code before any Android permission checks. Saint queries the AppPolicy provider for policies that match the IPC (step 2). The AppPolicy provider identifies the appropriate policies, checks that the policy conditions satisfied, and returns the result (step 3). If the conditions are not satisfied, the IPC is blocked; otherwise, the IPC is directed to the existing Android permission check enforcement software (step 4). Android will then allow (step 5) or disallow the IPC to continue based on traditional Android policy.

To protect both the interface use and the interface exposure, Saint enforces two types of runtime policies: 1) access policies identify the caller’s security requirements on the IPC, and 2) expose policies identify the callee’s security requirements on the IPC. That is, access policies govern the IPC an application initiates, and expose policies govern the IPC an application receives. Note that the target (for access) and source (for expose) are implicitly interpreted as the application specifying the policy, and an application cannot specify policy for other applications.

One can view Saint policy as being similar to a network-level stateful firewall [171]1. Like a stateful firewall, Saint identifies policy by its source and destination, and checks conditions to determine if the IPC should be allowed. In Saint, the source and destination are applications, components, Intent (event) types, or some combination thereof. Conditions are checks of the configuration or current state of the phone. Note that unlike a firewall, all Saint policies that match an IPC must be satisfied. Moreover, if no such policies exist, the IPC is implicitly allowed. Thus, from a technical standpoint, Saint is a “conjunctural default allow policy” rather than “default deny first match policy” [172].

4.3 Developing Application Security Policy

Saint provides an approach for the application developers to protect their own applications and their clients from potentially dangerous or compromised applications through interactions while being able to benefit from the converging application model. The understanding of the applications’ goals, functional requirements, and security concerns is essential for providing appropriate protection. This section presents the second prong of our methodology (see Section 3.3.1 for our three-prong methodology for mobile phone security policy development). We explore the use of security requirements engineering principle to solicit such knowledge from the developers and manipulate it to create the

---

1A stateful firewall maintains ordered policies of the type \(\{\text{source address, destination address, flags}\}\), where source and destination are IP address/ports pairs, and the flags represents the required state of the communication, e.g., whether a ongoing TCP connection between the source and destination exists.
application security policies. Based on the two main supports that Saint aims to provide as described in Section 4.2.2, our approach produces two types of policies.

**Install-time Permission Granting Policies** describe the conditions under which the permissions declared by the application can be granted to other applications when requested. This type of policies directly enhances the Android’s permission-based policy model, which grants permissions based only on their protection level. Being enforced at installation time, these policies do not induce run-time overhead. However, they help protecting the application only when it is being accessed but not when accessing others.

**Run-time Interaction Policies** regulate how the application interacts with other applications via run-time enforcement. They detail the conditions under which the application allows different types of interactions. The policies of both the caller and the callee must be satisfied for the interaction to occur. While this type of policies causes run-time overhead, it protects the applications both when they act as the callers and the callees.

Application security policies place constraints on (i.) other applications, i.e. the actors, and on (ii.) the operations. For permission granting policies, the operations being restricted are the grant of permissions. The constrained operations in the interaction policies are the interactions, which are determined by the type of functionality being requested for and the type of actions (e.g. start Activity, send Intent, etc.).

### 4.3.1 Application Security Policy Development Methodology

Unlike developing the platform security policies described in Section 3.3.2 which the functions of the applications are unknown to the platform developers and the users, the application developers know the functionality of the applications they create. This allows us to extract the application’s assets associated with each of its functions as introduced in the Problem Frame-based approach [102, 101, 103]. It is selected as a foundation of this prong of our approach because it is designed to develop security requirements for the systems during the system development process by looking at how the system interacts with other entities in the problem context. This complies with our scenario which the application developers build their security policies during the time the applications are developed. The policies constrain the applications’ interactions with other applications in the phone environment.

Nevertheless, there is still some slight deviation between our methodology and the Problem Frame-based approach. The Problem Frame-based approach aims at building appropriate security mechanisms into the system being developed to satisfy its own
security requirements. Most of these mechanisms require some participation from the external entities e.g. for authentication. This model does not suit the Android’s application model which aims to be seamless and involve minimum degree of coordination from other applications. To address this issue, our on-platform security solution [173] leverages the platform’s knowledge about the installed applications which can be used for authentication purposes. Our application security policies are to be enforced by the platform using the platform’s knowledge about other applications to make the security decisions. As a result, these policies must be enforceable by the platform, requiring only the information that is available to or obtainable by the platform.

This prong consists of five major steps as shown in Figure 4.3. It can be noticed that the procedure is nearly identical to the one used for platform policy development as presented in Section 3.3.2. However, the Step 1 and 2 are in reverse order due to the reason above. In addition, the detail for each step is also different.

**Step 1: Identify Application’s Functions**

To create the application security policies, we first enumerate the functions in the application which allow interactions with other applications. Both directions of the communication are considered, namely, when exposing its functionality to other applications and when requesting for functionality from others.

The problem diagrams such as in Figure 4.4 model functions including the sequences of communication and the information being passed between domains. Each domain is presented with a box. Boxes with two vertical lines imply the domains under the control
of the system developers. In our case, a domain represents an application. Dotted circles in problem diagrams denote functional requirements with the arrows pointing to the domains that must fulfill the requirements. These arrows inherently indicate the direction of the interactions. More specifically, the arrows pointing into the application depict the exposure of its functions while those pointing out represent the access to other applications. Based on this information, we create two separate lists. The *expose* list contains the exposing functions while the *access* list stores the accessing ones.

**Step 2: Identify Application’s Asset**

By examining the problem diagrams, we identify the assets associated with each of the functions. The asset can be the information being passed, the components, or mechanisms used to accomplish the functionality or are impacted by the functionality. When exposing its functions, the application’s assets normally include the functionality itself and the returned data. The assets when the application accesses other applications are usually the submitted data, the result, and its transaction that depends on the correctness and the availability of such functionality. Lastly, as we also take the users’ security and privacy concerns into account, what deemed important to them are also considered as the assets.

**Step 3: Determine Asset Security Goals and Threat**

This step is mostly identical to the Step 3 in Section 3.3.2. Our security goals are confidentiality, integrity, and availability. However, the possible threats depend on the types of assets, functionality, and the types and directions of the interactions.

When exposing their functionality, the threats usually involve the confidentiality of the returned data and the availability of the functions. However, the application developers know how the functions operate and should be able to judge how it can be potentially compromised by bad input and DoS attacks. On the other hand, when relying on the functionality from other applications, the applications normally concern about the threats relating to the integrity of the submitted data in addition to its confidentiality because it affects the correctness of the transactions. Moreover, another primary threat is the availability of the functions especially when the callers do not be notified of success or failure. Similar to Step 3 in Section 3.3.2, we use attack trees to model the potential threats and develop the threat descriptions.

**Step 4: Develop Application’s Security Policies**
We combine the threat descriptions from the Step 3 with our knowledge about the application’s functions in developing the policies. The functions indicate the operation parts of the policies while the threats depict what conditions must be met to prevent them.

For the functions in the expose list, the threats can be prevented using either the install-time policies or run-time policies. If the same conditions can be used for protecting several components and the phone’s context information is not required, using install-time policies is preferred because it does not cause run-time overhead. The required conditions are used to develop the policy for granting a particular permission. Then, the permission is used to protect those components but cannot be used to protect any other components with different exposure conditions. These conditions constrain on the properties of the applications that can hold the permission. The run-time policies can protect the functions in both the expose and access list. This type of policies describes the type of runtime interactions they protect and the properties of the applications allowed to interact with.

**Step 5: Determine Security Mechanism Limitations**

Similar to Step 5 in Section 3.3.2, we must refine the security policies such that they are enforceable by the platform. The run-time interaction policies can constrain the type of interactions only by two of their properties; namely, by the type of components to be interact with (e.g. Activity, Service, etc.) and by the action strings of the Intents used to initiate the interactions which is used by the Android platform to distinguish the interactions.

The types of conditions to be included in the security policies are limited to the information available to the Android platform. For instance, the identities of the applications can only be the hash of the application packages or the signatures on them. The capabilities and configurations of the applications are limited to those visible to Android such as the set of permissions they possess, their application versions, and the Intent messages they listen to. Lastly, the phone’s context can include only those available on the phone such as the phone’s locations, time, and call state.

**4.3.2 Sample Application Security Policy**

We consider the development of the policies for our PersonalShopper example presented in Section 4.2.1 to illustrate the use of our 5-five procedure. In developing the policy to control how the PersonalShopper interacts with the payment applications, we
start with describing such function in Step 1. The problem diagram for this function is shown in Figure 4.4. Only the interaction between the domains is presented. Here, \( a = APP!(\text{payment info.}) \), \( PAY!(\text{payment confirm}) \) depicts that when passing payment information to payment application, PersonalShopper shall receive the payment confirmation. The associated functional requirement shows that this is an access function.

In Step 2, one of the assets is the payment information, which is critical for the user, the merchant, and the financial institution that is responsible for the payment.

The security goals for this asset for the Step 3 include protecting its confidentiality, integrity, and availability. The threats to such goals can be (i.) malicious payment application reveals payment information to adversary, (ii.) malicious payment application alters the payment information without authorization, and (iii.) malicious payment application does not process the payment. Here, we present the attack tree of the threat (i.) in Figure 4.5. To achieve the attacker’s goal, the application must come from the adversary (2.A), connect to the Internet (2.B), and receive the payment information from the PersonalShopper (2.C). The latter two are required to perform the application’s functions because the payment application must process the payment through the Internet and will always receive the payment information. Therefore, the attacker goal is accomplished when 2.A is achieved.
In response to 2.A, the best way to protect the payment information from the above threats is to place the security policy in Step 4 that only the payment applications from companies with high reputation are trusted to receive the payment information and process the payment.

In the last step, we consider that the PersonalShopper would issue the Intent, e.g. `ACTION_PAY` to start the activity or to bind to the service offered by the supporting payment applications. The platform resolves the Intent to find the set of applications handling such Intent. In our running example, the action for the interaction can be `start activity with ACTION_PAY`. The developer must define the conditions for the trusted payment applications. An option is to supply a list of the payment applications that are trusted as the ground for his/her decision. It is only this set of payment applications that the PersonalShopper wants to interact with. To ensure the authenticity of these applications, they are identified by their developers’ signatures. Finally, we have the security policy of the PersonalShopper to interact with payment applications as For starting activity with ACTION_PAY, only the payment applications with signatures in the given set of trusted signatures S can interact with the PersonalShopper.

4.3.3 Application Security Policy Taxonomy

We explored a myriad of applications as a means of understanding the appropriate set of policy conditions which depicts the policy expressibility. An initial policy taxonomy is presented in Figure 4.6.

Figure 4.6. Policy tree illustrating the example policies that are required by applications. The boxed with double stroke represent the policies already supported by the original Android platform.
4.3.3.1 Install-time Permission Granting Policy

The permission-granting policy (1.) regulates permission assignment. In addition to controlling permission granting using Android’s protection level-based policy (1.1), an application $A$ may require *signature-based policy* (1.2) to control how the permissions it declares are granted based on the signature of the requesting application $B$ ($A$ and $B$ may be signed by different developer keys). Instead, the policy grants (or denies) the permission by default with an exception list that denies (grants) the applications signed by the listed keys. An application may also require *configuration-based policy* (1.3) to control permission assignment based on the configuration parameters of the requesting application, e.g., the set of requested permissions and application version.

As shown in Table 4.1, Saint install-time policy consists of a permission label, an owner, and a set of conditions. The permission label identifies the permission to be regulated. The owner is always the application declaring the permission. The conditions are a collection of checks on the properties of the application requesting for it. All checks must be true for the installation to be allowed. The condition can check the signatures on the application package or other permissions the application requests, i.e., the permissions it would possess if installed. The condition check is implicitly affirmative in that it requires the condition to be true, e.g., as the accepted developer signatures or required set of permissions. Alternatively, it can be negated e.g., as forbidden permissions. Only the application declaring such permission is allowed to create the policy for it.

The install-time policy for Requirement (5) of our motivating example in Section 4.2.1 is provided as policy (1) in Table 4.1. Saint encodes it in XML as:

```xml
<permission-grant permission="com.abc.perm.getloc"
    owner="com.abc.lbs">
    <required-permissions>
        <permission-label>
            android.permission.ACCESS_FINE_LOCATION
        </permission-label>
    </required-permissions>
</permission-grant>
```

4.3.3.2 Run-time Interaction Policy

The interaction policy (2.) regulates runtime interaction between an application and its opponent. An application $A$’s opponent is an application $B$ that accesses $A$’s resources or is the target of an action by $A$, depending on the access control rule (i.e., $B$ is $A$’s opponent for rules defined by $A$, and $A$ is $B$’s opponent for rules defined by $B$). Android’s
## Install-time policies:
(permissions-label) (owner) ![cond_1] ![cond_2] ![cond_n]

1. (com.abc.perm.getloc) (com.abc.lbs) required-permission(ACCESS_FINE_LOCATION)
   Permission com.abc.perm.getloc declared by com.abc.lbs only be granted to applications with ACCESS_FINE_LOCATION permission

2. (com.me.perm.getrec) (com.me.voicerecord) forbidden-permission(INTERNET)
   Permission com.me.perm.getrec declared by com.me.voicerecord only be granted to applications without INTERNET permission

## Run-time policies:
(expose|access) (source app, type, action) (destination app, component) ![cond_1] ![cond_2] ![cond_n]

3. (access) (com.ok.shopper, START_ACT, ACTION_PAY) (any, any) sig:default-deny:except(3082019f3082 . . .)
   com.ok.shopper cannot start activity with ACTION_PAY action to any components in any applications unless they have signature 3082019f3082 . . .

4. (access) (com.ok.shopper, any, any) (com.secure.passwordvault, any) min-version(1.2)
   com.ok.shopper can start any interaction with any action to any component in com.secure.passwordvault version 1.2 or higher

5. (access) (com.ok.shopper, any, RECORD_EXPENSE) (any, any) forbid-permissions(INTERNET)
   com.ok.shopper cannot start any interaction with action “RECORD_EXPENSE” to any component in any application with permission “INTERNET”

6. (expose) (any, any, any) (com.trustbank.adminconsole, any) loc:default-deny:except((39.623616,-77.411827,100),(39.425052,-77.415765,100))
   Any application cannot start any interaction with any action to any component in com.trustbank.adminconsole unless the phone is within 100 meters radius from (39.623616,-77.411827) or 100 meters from (39.425052,-77.415765)

### Table 4.1. Sample security policies for secure application interactions

Existing permission-based access control policy (2.1) provides straightforward static policy protection, as described in Section 2.2.3. However, this policy is coarse-grained and insufficient in many circumstances. Applications may require signature-based policy (2.2) to restrict the set of the opponent applications based on their signatures. Similar to above, the default-allow and default-deny modes are needed. With configuration-based policy (2.3), the applications can define the desirable configurations of the opponent applications; for example, the minimum version and a set of permissions that the opponent is allowed (or disallowed). Lastly, the applications may wish to regulate the interactions based on the transient state of the phone. The phone context-based policy (2.4) governs
runtime interactions based on context such as location, time, Bluetooth connection and connected devices, call state, data state, data connection network, and battery level. Note that initially, policy types 2.2 and 2.3 may appear identical to 1.2 and 1.3; however, the former types also place requirements on the target application, which cannot be expressed with 1.2 and 1.3. However, 1.2 and 1.3 are desirable, because when applicable, they have insignificant runtime overhead.

As shown in Table 4.1, Saint runtime policy consists of a type label, source application details, destination application details, and a set of conditions. The expose/access label identifies the policy type. Applications must only govern their own IPC, therefore, the specifying application must be the destination for access polices and source for expose polices. The source identifies the caller application and, if applicable, the Intent. The definition of a destination callee of a policy is somewhat more flexible.

The destination can be an application, a component, an Intent, or an application/Intent combination. We expand the notion of destination to provide finer granularity policy, as applications with many interfaces frequently require per-interface policies. For example, the security policy governing an “add an item to my shopping cart” feature provided by one component may be very different than the “authorize this transaction” component policy.

Runtime policy rules specifying multiple conditions require all conditions to be true for the IPC to proceed. Conditions can be any test that returns a Boolean value. For example, test for permission configuration, roaming state, or any other evaluation function the application deems necessary. Conditions may also be negated, indicating the IPC should only proceed when the condition is not satisfied (e.g., to blacklist configurations). For example, a reasonable policy might prevent the phone’s web browser from accessing the address book or dialer.

Policy (3), (4), and (5) in Table 4.1 provide runtime policies in response to security requirements (1), (3), and (4) of the example in Section 4.2.1. In Saint, runtime policy (3) is presented in XML as:

```xml
<interaction direction="access">
  <source>
    <application>com.ok.shopper</application>
    <interaction-type name="START_ACTIVITY" />
    <action>ACTION_PAY</action>
  </source>
  <destination>
    <application>any</application>
  </destination>
  <condition>
</interaction>
```
4.4 Auxiliary Policies

In spite of the application policies that regulate the applications’ behaviors, we also need other types of policy to facilitate our administrative tasks.

4.4.1 Administrative Policy

An administrative policy dictates how policy itself can be changed [10]. Saint’s default administrative policy attempts to retain the best qualities of mandatory access control in Android: all application policies are fixed at installation and can only change through application update (reinstallation). In Saint, the application update process removes all the of relevant policies and inserts those specified in the update. From a policy perspective, this is semantically equivalent to uninstalling and installing an application. We considered other administrative models allowing the updater to modify, add, or delete policy. However, the phone policy could unpredictably diverge from that desired by the developer quickly where, for example, update versions were skipped by the user.

There is a heated debate in smartphone operating systems community about whether to allow users to override system/application policies. A purist school of thought suggests that applications are providing MAC policies, and therefore, nothing should be changed. This provides the most stringent (and predictable) security, but potentially can prevent otherwise legitimate operations from occurring. The second school of thought says the user is always right and every policy should be overridable.

There is no one right answer to the override debate. Hence, we introduce an infrastructure for overriding, but leave it as an OS build option. If the SaintOverride compile flag is set, Saint allows user override to application policy. Additionally, Saint XML policy schema includes the Override flag for each policy rule defined by the application. If the system SaintOverride system flag and Override flags are true, the FrameworkPolicyManager application (see Section 4.5) allows the user to disable the rule though the interface. If disabled, this rule is ignored during policies decisions. Note that we considered allowing the user to arbitrarily modify the policy (rather than simply disabling it), but this introduces a number of complex security and usability concerns
that we defer to future work.

4.4.2 Operational Policy

Saint has the potential to hamper utility by restricting access to interfaces. Detecting such incidents is essential to be providing a useful service. Past security measures that have prevented application behavior in an opaque and ambiguous way have not fared well, e.g., Microsoft Vista. This section defines policies that detect when Saint renders an application inefficient, faulty, or inoperable.

Consider a simple logical formulation of the Saint runtime policies. The conditions supported in the system are denoted by the set $C = \{c_1, c_2, \ldots, c_n\}$. $C$ can be further subdivided into two sets $V$ and $T$, i.e., $C = V \cup T$. $V$ is the set of conditions which are invariant with respect to the system state. Invariant conditions do not change as a function of the normal operation of the phone. For example, permission assignments, developer signatures, and application version numbers are invariant. $T$ is the set of conditions which rely on transient system state, e.g., roaming state, battery power, and access to a 3G interface.

Recall that IPC is governed by the access policy $p_a$ of the caller and the expose policy $p_e$ of callee. A given interaction will succeed only if the conditions of both policies are satisfied. Logically speaking, each policy consists of zero or more elements of $C$ or their negation. At any given time, the system state of the phone $S$ is a truth assignment for Boolean variables for each element of $C$. $\hat{S}$ is the set of all possible sets of $S$. Let $V$ be the subset of $S$ relating to elements of $V$ (the invariant conditions). The run-time IPC decision is therefore a simple test of satisfaction of the conjunction of $p_a$ and $p_e$ by $S$, i.e., $S \Rightarrow p_a \land p_e$.

This formulation allows us to reason about the satisfiability of policy at install time. There are three possible outcomes for the install-time analysis of future IPC:

- $V \Rightarrow p_a \land p_e$ (always satisfied)
- $\exists S \in \hat{S} \mid S \Rightarrow p_a \land p_e$ (satisfiable)
- $\not\exists S \in \hat{S} \mid S \Rightarrow p_a \land p_e$ (unsatisfiable)

where, “always satisfied” IPC will always be able to proceed (because the invariant conditions never change), “satisfiable” can occur under certain conditions (because they depend on changing system state), and “unsatisfiable” will never be allowed to occur.

---

2Interfaces unprotected Saint policies are in essence “empty” policies. For the purposes of the logical analysis presented in this section, WLOG, they can be modeled simply by the Boolean value $TRUE$. 
This last case occurs when either rule contains an unsatisfied invariant condition, e.g., incorrect developer signature, or the two rules conflict, e.g., where the expose/access rule contains a condition $c$ and the other contains its negation $\neg c$. Note that because of the structure of the logical expressions, this satisfiability test can be tested in polynomial time.

We exploit this analysis to learn about the ability of an application to function. Saint tests every access rule of an application during its installation. Any rule that is unsatisfiable depicts an unusable interface, which may represent a serious functional limitation, e.g., imagine a text message application that cannot use the address book. Thus, the framework warns the user if any access rule is unsatisfiable. Moreover, we add FeatureRequirement enumerated value to the XML structure of each policy rule. This value has three values; NONE, AVAILABLE, and ALWAYS. The NONE has no effect. The framework prevents the application from being installed if AVAILABLE is declared and the rule is unsatisfiable or if ALWAYS is declared and the rule is not always satisfied.

The operational policy allows the system to track and manage dependencies between applications and interfaces. By checking the operational policies of all applications during installation, update and uninstallation, we can detect when a change in an application will effect other applications. The system can warn the user or simply prevent the operation from moving forward if required interfaces become non-functional or are removed.

### 4.5 Implementing Saint Framework

Saint was implemented as a modification to the Android 2.1 OS. For each of the above install-time and runtime policies, we inserted one or more enforcement hooks into Android’s middleware layer. In this section, we describe the relevant functionality in Android and the modifications we made to enforce Saint policies.

#### 4.5.1 Saint Installer

The Saint installer is a modified version of Android’s application installer. The installer receives the path to the downloaded Android package (.apk) and parses the package using PackageParser and PackageManager. During this step, we collect all package configurations necessary for install-time policy evaluation, such as the package’s signature, requested permissions, and application version. The package’s declared permissions are also acquired to verify this package’s application policy.
We implement Saint’s policy in a separate XML file with name identical to the package name. We chose to express the application policy in XML to match the format of Android’s manifest file. Including the policy into the manifest file requires changes to the Android SDK and to the installer’s package parsing function. We consider this extension as our future work.

Immediately after the package is parsed, the Saint installer examines each requested permission against its corresponding permission-granting policy queried from the App-Policy provider. If a conflict is found, the installer rejects the installation.

After successful installation, the installer parses the application’s policy file to an intermediate form. By considering the application’s declared permissions obtained during the package parsing step, the installer ensures that each policy entry is inserted into the AppPolicy provider only if its permission label is declared by the application.

### 4.5.2 Saint Mediator

Saint’s runtime enforcement covers four critical component interactions: starting new Activities, binding components to Services, receiving broadcast Intents and accessing Content Providers. For each of these interactions, we cover the limitations of the existing Android security implementation and explain the necessary modifications and authorization hooks needed to enforce Saint policies.

**Starting Activities (4.A)** — As users interact with activities, they often spawn new activities for GUI elements such as menus and dialogs. In Android, a request to start a new activity takes the form of an Intent sent to the *Activity Manager Service (AMS)*, a
key Android component that facilitates interactions between activities.

The AMS will then match one or more activities that have registered for that Intent. In the event that a single match is not found, i.e. there are multiple registered activities, the list of all such activities is displayed to the user who chooses the correct one, e.g. should a photograph be sent to an email client or an album. When the destination activity is known, the AMS will check if the sending activity has the permission to start such activity. If so, the activity is started. This possibility for multiple activities to match an Intent represents one of the limitations of the current Android security framework in that the registered activity has no control what component may call it beyond the permissions needed for its Intent. The calling activity has no control over which target activity is selected. To allow both the source as well as the receiver activity to influence the decision to spawn the receiver, we add a hook that restricts the set of candidate activities to choose from as shown in Figure 4.7.

**Saint Hook Placement:** If a single activity matches the Intent when it is resolved by the AMS, hook (1) checks that the conditions for both the source and destination activity before starting the destination activity as a match for the Intent. If multiple activities are registered for the Intent, it is passed to ResolverActivity for further Intent resolution. For each of the matched activities, hook (2) checks the source against each potential destination before allowing it to be included in the list of user options. Any destination activities not allowed by the current policy are excluded from the list. The activity selected by the user is the target activity for the Intent. There is also a small probability that only one matched activity is found. This match is checked by hook (3) whether it can be the target. Then, the target activity is started through the AMS. This time, the Intent is addressed to the specific activity and will have only a single match. The final check is performed by hook (1) to prevent TOCTTOC attack.

**Receiving Intent Broadcasts (4.B)** — A Broadcast Receiver acts as a mailbox for the application. It listens to Intent message broadcast by another component in the same or different application for data exchange. To specify the type of messages it is listening to, the Broadcast Receiver is attached with Intent-filter(s) that describe Intent values to be matched including the action string. Intent broadcasts are handled by the AMS, which attempts to resolve the broadcast receiver components registered for the Intent. A broadcast receiver may be registered for receiving specific Intent(s) either statically at install-time or dynamically during its execution. A static Broadcast Receiver and its permanently associated Intent-filter(s) are declared in the manifest and is always registered with the system. In contrast, a dynamic Broadcast Receiver is declared as a
class in the code and is instantiated during runtime. It can be registered and unregistered any time. The Intent-filter(s) attached to the dynamic Broadcast Receiver is also created at runtime, thus can be changed.

Saint Hook Placement: In order to enforce Saint’s access policies for Intent broadcasts, several authorization hooks were inserted into this process. Hook (4) is taken if the broadcast receiver is selected by name. In this case, only a single check is performed for the named receiver. If the Intent contains an action string, it can be received by potentially multiple broadcast receivers. In this case, hooks (5) and (6) iterate over the lists of potential receivers and perform a policy check for each one before it is allowed to receive a message.

Accessing Content Providers (4.C) — In Android, applications access content providers by a URI. The Content Resolver is responsible for mapping a URI to a specific content provider and obtaining the IPC interface to the content provider that performs the operations (e.g. query, update, etc.). Android’s permission check is performed by the content provider during the operation execution. This check is inadequate to protect applications from a potentially malicious content provider that has registered under a particular URI.

Saint Hook Placement: To extend the enforcement to allow the source component to be protected as well, Saint places authorization hook (7) at the Content Resolver. The list of registered content providers is stored by the AMS in form of Provider Record. Therefore, our modified AMS provides the Provider Record that matches the authority string to the Content Resolver. The record contains information that allows application policy checking.

Binding Components to Services (4.D) — The last type of interaction mediated by Saint is binding a component to a service (allowing the component to access the service’s APIs). A component binds to a service either by specifying its name or an Intent containing an action string to which that service has registered. Binding to services is mediated by the AMS, which first resolves the service by name or action string and then checks that the requesting component has the necessary permissions to bind it.

Saint Hook Placement: We inserted a single mediation point, (8), into the AMS to check Saint policy before the Android permission check. Since access policies require the source component name in the hook, we extracted the source name from a field in the binding request. For the other types of component interactions where the source name was not available, we modified the execution path up to the hook to propagate the name of the
component initiating the interaction.

4.5.3  AppPolicy Provider

The policies for both the install-time and run-time mediator are stored in the AppPolicy provider. We embedded the AppPolicy provider inside the middleware in a way similar to the phone’s address book, calendar, and DRM provider are included in the platform. The policy is stored in SQLite database, which is the default database supported by Android. The database files for the provider are located in the system directory, e.g., in the /data/system/ directory.

More importantly, the AppPolicy provider is the policy decision point. At install-time, the Saint Installer passes the information about the package being installed to the AppPolicy provider for the decision making using the exposed verifyPermissionGrant API. The new policy is inserted using insertApplicationPolicy API. Both API interfaces are implemented as part of Android’s Activity API. At run-time, inside the middleware, the Saint mediator’s hooks consult the AppPolicy provider for policy decision based on the information about the source and the destination of the interaction.

To make the decision, the AppPolicy provider retrieves all matched policies and collects all information needed to evaluate the conditions. For interaction policy, it may need to contact Package Manager and several system services such as Location Service and Telephony Service, which require the caller to run under an application environment; thus cannot be accessed by the AppPolicy provider. To address the problem, we added more functions to the AMS, which runs under “android” application environment, to obtain the information for the AppPolicy provider.

Note that it is essential to protect the API interfaces for accessing the AppPolicy provider from malicious applications. If not protected, a malicious application could simply insert bogus policies to block legitimate IPC or delete others. The current AppPolicy provider checks the identity of the application that makes the API call. If the application is not the Saint installer, the request is denied. We foresee that it may be desirable for future applications of Saint to allow other applications to view policy (e.g., policy viewers, system diagnostics). The current system will be modified to either whitelist read, write, or delete for given applications or simply check to see they have received some other system Saint policy permission.
4.5.4 FrameworkPolicyManager

As mentioned in Section 4.4.1, FrameworkPolicyManager is implemented as an Android application to enable the user to override the policy if its override flag and the system's SaintOverride flag are true. It updates the policies in AppPolicy provider using updateApplicationPolicy API implemented in Android's Activity API. To prevent malicious applications from updating the policies, the identity of the application is checked to ensure that only the FrameworkPolicyManager can update the AppPolicy provider.

4.5.5 Condition Extensibility

So far, we have covered a set of policy enforcement mechanisms implemented by Saint. These policies are made up of conditions based on application configuration and phone state. Each condition requires code be run to inspect some aspect of either an application’s context or the device’s state. Currently, the AppPolicy provider is limited to the static set of implemented conditions covered in Section 4.3.3. Because we cannot predict the types of conditions future smartphone apps may wish to check in their security policies, Saint contains a generic mechanism to perform custom condition checks implemented by application developers. This mechanism works as follows.

At install time, an application’s package is checked for one or more ConditionCheckImpl classes. These classes are instantiated and registered by Saint at boot time. The application includes conditions in its runtime policy that are handled by its ConditionCheckImpl. Any time a component from that application is either a source or destination of one of Saint’s mediated component interactions, the condition check method of its ConditionCheckImpl class is called and the result is composed with the results of the Saint enforced conditions to make a policy decision. This method has the following signature: boolean checkCondition(String condition), where condition is a custom condition string provided in the application’s runtime policy and the return value is the result of the condition check.

4.6 Evaluating Applications Protection

Saint provides valuable enforcement for Android application providers. This section evaluates Saint in two perspectives. First, we demonstrate Saint’s value-add by discussing its applicability to applications associated with the OpenIntents collaborative project [174]. Second, we measure Saint’s runtime performance overhead.
4.6.1 Policy Appropriateness

Recall that Android applications primarily interact via Intent messages. The OpenIntents project exists to “…collect, design, and implement open intents and interfaces to make Android mobile applications work more closely together.” [174]. In so doing, OpenIntents provides a registry of Intent action strings standardized by both the Android framework and participating OpenIntents application providers. Applications part of the OpenIntents project use these action strings to handle events and provide services in a consistent manner, thereby providing value-added services to one another without requiring needless reimplementation of functionality between applications. By considering the collaboration and interaction between these applications, we demonstrate the need for the Saint policy framework by existing application providers.

Table 4.2 lists sample OpenIntents Intent action strings with corresponding service provider and consumer applications (a March 2010 survey of the OpenIntents project revealed 55 unique Intent action strings and 31 service provider and consumer applications). For ease of exposition, we use abridged action string names (e.g., SHOW_RADAR) instead of the full namespace. In all cases shown in Table 4.2, service providers and consumers are Activities contained by the listed application. Hence, a service provider is an application that handles an Intent action. For example, the Radar application contains an Activity that shows a radar graphical display upon handling the SHOW_RADAR action. Similarly, a service consumer is an application that makes use of functionality implemented by a service provider. For example, the Panoramio application uses the Radar application to display the user’s proximity to a landmark. Note that we describe these interactions from the perspective of service functionality and not the underlying mechanism, i.e., the corresponding service provider Activities “consume” Intent action strings. To avoid confusion, we always discuss functionality from the service functionality perspective.

The table enumerates a variety of service types: financial, location-based, cryptographic, and search functionality. In the provided examples, the Intent action string corresponds to this functionality. The listed service provider and consumer applications provide use cases upon which one can discuss threat models and enforcement policy. The NEW_TRANSACTION action string handled by the Funky Expenses application provides financial ledger functionality similar to the PersonalShopper scenario. In this case, the service provider is trusted with financial data. The SHOW_RADAR action string handled by the Radar application (mentioned above) provides location visualization for geo-tagging applications. In this case, the service provider is trusted with location information. The
<table>
<thead>
<tr>
<th>Intents</th>
<th>Important Extras</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) com.funkyandroid.action.NEW_TRANSACTION: This intent reports a financial transaction to be recorded in an expenses or banking application</td>
<td><strong>Input:</strong></td>
<td>Service Providers:</td>
</tr>
<tr>
<td></td>
<td>- Payee for a transaction</td>
<td>- Funky Expenses</td>
</tr>
<tr>
<td></td>
<td>- Transaction amount</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Category of the item</td>
<td></td>
</tr>
<tr>
<td>(2) com.google.android.radar.SHOW_RADAR: Display a radar like view centered around the current position and mark the given location</td>
<td><strong>Input:</strong></td>
<td>Service Providers:</td>
</tr>
<tr>
<td></td>
<td>- latitude</td>
<td>- Radar</td>
</tr>
<tr>
<td></td>
<td>- longitude</td>
<td>- Panoramio</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Photostream</td>
</tr>
<tr>
<td>(3) org.openintents.action.ENCRYPT: Data encryption functionality which is protected by org.openintents.safe.ACCESS_INTENTS dangerous permission.</td>
<td><strong>Input:</strong></td>
<td>Service Providers:</td>
</tr>
<tr>
<td></td>
<td>- URI to the file or text data to be encrypted</td>
<td>- OI Safe</td>
</tr>
<tr>
<td></td>
<td>- URI to encrypted file or encrypted data</td>
<td>- Obscura</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- MyBackup Pro</td>
</tr>
<tr>
<td>(4) org.openintents.action.DECRYPT: Data decryption functionality which is protected by org.openintents.safe.ACCESS_INTENTS dangerous permission.</td>
<td><strong>Input:</strong></td>
<td>Service Providers:</td>
</tr>
<tr>
<td></td>
<td>- URI to the file or text data to be decrypted</td>
<td>- OI Safe</td>
</tr>
<tr>
<td></td>
<td>- URI to decrypted file or decrypted data</td>
<td>- Obscura</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- MyBackup Pro</td>
</tr>
<tr>
<td>(5) org.openintents.action.PICK_FILE: Pick a file through a file manager.</td>
<td><strong>Input:</strong></td>
<td>Service Providers:</td>
</tr>
<tr>
<td></td>
<td>- Title of text to be displayed</td>
<td>- OI File Manager</td>
</tr>
<tr>
<td></td>
<td>- Text on the button to be displayed</td>
<td>- Convert CSV</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Notepad</td>
</tr>
<tr>
<td></td>
<td>File URI of the selected file</td>
<td></td>
</tr>
<tr>
<td>(6) android.intent.action.SEARCH: Perform a search</td>
<td><strong>Input:</strong></td>
<td>Service Providers:</td>
</tr>
<tr>
<td></td>
<td>- Query string for search action</td>
<td>- Collectionista</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- WikiNotes</td>
</tr>
</tbody>
</table>

Table 4.2. Sample OpenIntents Intent action strings with service provider and consumer applications.

The `ENCRYPT` and `DECRYPT` action string handled by the OI Safe application provide cryptographic functionality. In these cases, the service provider is trusted with sensitive data, and it must trust service consumers with sensitive data it returns. For example, OI Safe is trusted not to release data encrypted by the OI Notepad application to the Obscura application. The `PICK_FILE` action string handled by the OI File Manager application allows the user to graphically search for a data file on SDcard. In this case, the service provider is trusted to safely access the user’s data and directory structure on the SDcard.
and return the correct file selected by the user. Finally, the SEARCH action string handled by the Collectionista and WikiNotes applications for domain specific content search. In this case, the service providers are trusted to uphold privacy agreements made with the user.

We now demonstrate how Saint policies provide protection not expressible within Android’s existing framework. We separate corresponding policies into classes based on security goals.

**Policy to Prevent Information Exposure:** Service consumers frequently trust that service provider applications do not expose sensitive information. Frequently, the information exposure perimeter is the phone itself. That is, the information should not leave the phone. The following example policy protects the service consumer by ensuring the service provider does not have Internet access.

```
“consuming-application cannot start any interaction with action NEW_TRANSACTION to any component in any application with permission INTERNET”.
```

Saint’s information exposure policies are not limited to Internet permission checks. Saint also provides methods of whitelisting and blacklisting applications based on various features, e.g., cryptographic signature and version number. For instance, any applications including the malicious ones can offer encryption and decryption functionality. The service consumer must prevent sending sensitive information to the malicious applications for encryption. To prevent this, it can specify that it uses only such functions offered by the applications from the developers it trusts. This requirement can be specified in Saint policy by including the signatures on the IO Safe application (which identifies its developer) in the set of trusted signatures. Observably, Saint provides security while preserving the flexibility of OpenIntents since the signatures on other trusted applications for encryption/decryption operations can also be included in the set. The Saint policy satisfying this requirement is as follow.

```
“consuming-application cannot start activity with action ENCRYPT to any component in any application unless it has signature included in trust-signature-set.”
```

**Policy to Prevent Information Misuse:** An application provider may also wish to ensure it does not inadvertently misuse information. For example, consider the SHOW_
RADAR Intent action string. The Intent takes latitude and longitude information as input. The application provider may wish to ensure that it does not misuse location information by passing it to an application that does not have permission to read location information. The following example policy protects against such a misuse.

“consumer-application cannot start activity with action SHOW_RADAR to any component in any application without permission ACCESS_FINE_LOCATION”.

Policy to Ensure Separation of Duty: Android allows third-party applications to implement and extend certain “system”-level functionality. To do so, Android assigns associated permissions with the dangerous protection level. While users may desire such applications to be installed, application providers (and users) may desire a separation of duty when initiating communication with a service provider. The following example ensures that NEW_TRANSACTION service provider does not have the SET_PREFERRED_APPLICATIONS permission, which allows the service provider to change the default handler without the user’s knowledge.

“consumer-application cannot start any interaction with action NEW_TRANSACTION to any component in any application with permission SET_PREFERRED_APPLICATIONS”

Policy to Protect Access to Functions: Occasionally, service provider applications want to restrict access to their functionality. As an example, OI Safe application declares ACCESS_INTENT permission which by its description allows the applications to encrypt and decrypt text, and access the passwords they have stored in OI Safe. In other words, the service consumer applications need to hold this permission to use such functions. However, the ACCESS_INTENT permission has dangerous protection level which can be generously granted by the user. Allowing malicious applications to have this permission and gain access to such functionality can introduce vulnerability. For instance, the malicious applications can issue get password requests. While access to the passwords is allowed only if the requesting applications’ package names are included in the passwords’ allow lists, the package names are not good authentication tokens. A malicious application can name itself as another legitimate package. It can be installed if the legitimate one is not installed on the phone. As a result, it can potentially access the password of some other applications. Saint policy can provide better protection by allowing the
OI Safe application to place more restrictions on how the ACCESS_INTENT permission is granted. For instance, it can specify in the Saint policy the set of developers it trusts to use its functions as identified by the developer keys used to sign their applications. As an example, since OI Safe trusts OI Notepad, Obscura, and MyBackup Pro to use its functions, it includes the signatures on these applications in its set of trusted signature. This permission granting policy is expressed as:

"application cannot be granted with ACCESS_INTENT permission unless it has signature included in trust-signature-set."

**Policy to Ensure Correct Service:** Applications use utility functions offered by other applications to assist their operations. For example, service consumer applications select their files using OI File Manager. Similarly, they use search functions provided by Collectionista. In these scenarios, the service consumers must be able to trust that the returned and displayed results are correct. An alternative is to ensure that they actually use the functions from the trusted service provider applications. Similar to the prevention of information exposure, the set of trusted sources of applications can be identified by the developer keys used to sign the applications. The trusted developers of the applications for selecting file can be Google and the developers of OI File Manager and Astro File Manager, while for searching can be Google and the developers of Collectionista and OpenSearch. The signatures of the trusted developers for each action can be specified in the policy associated with it. The policies for PICK_FILE and SEARCH actions can be:

"consuming-application cannot start activity with action PICK_FILE to any component in any application unless it has signature included in trust-signature-set-for-select-file."

"consuming-application cannot start activity with action SEARCH to any component in any application unless it has signature included in trust-signature-set-for-search."

### 4.6.2 Performance Evaluation

We now consider the implications of Saint’s runtime policy enforcement for performance. This is done at two time-scales. Macrobenchmarks are performed under different policy configurations to capture the human time-scale impact of Saint mediation. Microbenchmarks are used to show the overhead created by both the policy database access and the
Table 4.3. Example Scenarios for Saint’s Performance Overhead

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Without Saint</th>
<th>With Saint</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Activity</td>
<td>533.69 ms</td>
<td>562.83 ms</td>
<td>29.14 ms</td>
</tr>
<tr>
<td>Start Activity for Result</td>
<td>477.20 ms</td>
<td>506.73 ms</td>
<td>29.53 ms</td>
</tr>
<tr>
<td>Broadcast Intent</td>
<td>306.51 ms</td>
<td>335.96 ms</td>
<td>29.45 ms</td>
</tr>
<tr>
<td>Bind Service</td>
<td>284.48 ms</td>
<td>314.26 ms</td>
<td>29.78 ms</td>
</tr>
<tr>
<td>Access Content Provider</td>
<td>293.55 ms</td>
<td>323.21 ms</td>
<td>29.66 ms</td>
</tr>
</tbody>
</table>

phone state checks. All experiments were repeated until the 95% confidence interval was less than two orders of magnitude of the mean. It is worth mentioning for the sake of completeness that we used third party software to increase the amount of available storage on the G1 [175] and to maintain compatibility with the Android 2.1 platform [29]. Neither of these impact Saint’s performance.

4.6.2.1 Macrobenchmarks

In this section, we measure the effects of Saint on each of the four mediated operations: Activity creation, Intent broadcast, Service binding and Content Provider access. We use a simple in-house benchmark that exercises each operation a minimum of 120 times. The benchmark was run with Saint configured for the developer signature test policy described in section 4.3.3. During the experiment, the phone is set to stay awake to keep the CPU on.

Table 4.3 shows the execution time of the scenarios when an Activity in a caller application performs five main types of interaction with a callee application. There are 20 Saint policies of different types on the system. Both the caller and the callee application have their policies restricting the accepted signatures.

It can be observed that the runtime overhead of Saint is slightly under 30 ms which accounts for 5-10% of the execution time. We consider this overhead acceptable because it should not be detected by human users and only occurs during cross components interaction. Note that for both types of starting Activity, our execution time excludes the time the screen window is composed and set visible. As a result, our reported time (for both the system with and without Saint) is slightly smaller than the time reported by the Android’s Logcat tool.
4.6.2.2 Microbenchmarks

We measure the execution time of major Saint operations. Using the experiment setup in Section 4.6.2.1, Figure 4.8 details the underlying performance overhead of Saint for each of the interaction scenarios in Table 4.3. Most of the cost results from SQLite database operations, namely database query (≈11.58 ms), record reading (≈10.67 ms), and database cursor operations (≈5.741 ms). While more than half of the cost is fixed, some overhead varies according to the number of policies to be evaluated. In our setup, there are two matched policy records in the database; each requires checking of signatures on the application. The variable cost is approximately 6 ms per matched policy.

Table 4.4 shows the execution time for retrieving the applications’ information and the phone context which are used for Saint policy evaluation. The latency from obtaining the application configuration is insignificant. On the other hand, the retrieval of the phone’s current context accounts for some performance overhead.

Lastly, to investigate the scalability of our AppPolicy database, we experimented Start Activity interaction under the settings of different sizes of AppPolicy database. As shown in Figure 4.9, the overhead from database query is relatively stable at under 12 ms regardless of the number of Saint policies in the database.

4.7 Summary

The evolving mobile ecosystem has changed the way mobile phone applications operate. The more open and converging application model allows applications to expose their
<table>
<thead>
<tr>
<th>Information</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application Information</strong></td>
<td></td>
</tr>
<tr>
<td>Requested Permissions</td>
<td>0.330 ms</td>
</tr>
<tr>
<td>Signatures</td>
<td>0.4 ms</td>
</tr>
<tr>
<td><strong>Context Information</strong></td>
<td></td>
</tr>
<tr>
<td>Data Connection Type</td>
<td>0.233 ms</td>
</tr>
<tr>
<td>Call State</td>
<td>3.333 ms</td>
</tr>
<tr>
<td>Data State</td>
<td>3.433 ms</td>
</tr>
<tr>
<td>Roaming State</td>
<td>0.311 ms</td>
</tr>
<tr>
<td>Location</td>
<td>2.967 ms</td>
</tr>
<tr>
<td>Time</td>
<td>14.4 ms</td>
</tr>
</tbody>
</table>

Table 4.4. Performance overhead of obtaining information for policy evaluation

![Figure 4.9. AppPolicy database query overhead](image)

internal functions to other applications and use those provided by others. While promoting innovation, this application model enables malicious or compromised applications to pollute healthy applications.

This chapter approaches mobile phone security from the viewpoint of mobile phone applications in relation to such application interactions. We proposed Saint framework which addresses the current limitations of the Android security model in protecting the applications. Saint adopts security requirements engineering to building application security policies, namely install-time permission granting policies and run-time inter-application communication policies, which reflect the application developers’ requirements, and provides appropriate policy enforcement mechanisms. We also provide operational policies to expose the impact of security policy on application functionality, and to manage dependencies between interfaces. Driven by the analysis of many applications, our investigations have provided an initial taxonomy of relevant security contexts.
Lastly, we show the usefulness of Saint through the case study on OpenIntents project and show that Saint incurs only small performance overhead thus is practical for real world application.

We seek to extend the Saint policies to protect the phone “system” services and the cellular network, as well as integrate its interfaces with widely used security infrastructures, e.g., PKIs, enterprise systems, etc. Moreover, the current Saint framework only protects overt communication through the system-defined APIs. An interesting future work is to investigate an approach to address cross-application communication via covert channels. Through ongoing feature enhancement and user study, we hope to transition Saint from a research system to a viable framework for the many millions of phones that will soon run Android.
Chapter 5

Secure Handling of Mobile Content

The revolution in mobile phone applications does not only drive communication among applications on mobile phones but also extends beyond the phone boundaries to remote parties external to the phones. Users now commonly exchange various forms of mobile content ranging from SMS, MMS, instant messages, emails, to data from Internet connections, when exchanging personal messages and photos, playing games, browsing websites, and viewing multimedia. Such mobile content comes from different sources such as other phone users, application providers, financial institutions, and enterprises. From the angle of the phones, these senders can be viewed as external content providers. Akin to other stakeholders in the mobile ecosystem, these external content providers also have certain expectations on the behaviors of the phones. More specifically, they have their own security requirements regarding how the phones would treat each piece of their data. Nevertheless, our understanding of these requirements and how they can be satisfied is still very limited.

Current phone platforms provide protections for user privacy, the cellular radio, and the integrity of the OS itself. However, few offer protections to protect the content. Once it enters a phone it is completely under control of the phone platform and the applications that initially receive it thus rendering sensitive documents vulnerable to access by arbitrary applications.

Open platforms such as Android provide few direct protections for the content placed on the phone. Access controls restrict access to application interfaces (e.g., by placing permissions on application components in Android), rather than placing explicit access controls on data they handle. Therefore, what limited content protections exist are largely a by-product of the way interfaces are designed and permissions (often capri-
ciously) assigned. Thus, a malicious application with the appropriate permissions can exfiltrate even the most sensitive of data from the phone. Malware has recently begun to exploit such limitations [176, 177, 178]. Moreover – even in the absence of malicious applications – commercial interests such as media providers wish to provide content without exposing themselves to content piracy.

This chapter aims to approach secure mobile phones from the perspective of the mobile content. We explore classes of security requirements and seek for pertinent technique to deliver digital rights management (DRM) policy and enforce it on smart phones. An analysis of the Android market shows that DRM services should ensure: a) protected content is accessible only by authorized phones b) content is only accessible by provider-endorsed applications, and c) access is regulated by contextual constraints, e.g., used for a limited time, a maximum number of viewings, etc.

Implementing this concept for every type of content at once is far-fetched because each of them is delivered and handled differently. In our pilot study, we illustrate the expressibility and enforcement of our proposed content policies in controlling content obtained over SMS, MMS, and email.

The chapter begins with an overview of our environment and problem scope. Next, we bring out the limitations of the current mobile content handling regime particularly, OMA DRM, a prevalent standard for content protection on mobile devices, and therefore propose Porscha, Policy ORiented Secure Content HA ndling framework that enforces DRM policies embedded in received content. By imposing provider specified policies, Porscha enables senders of SMS, MMS, and email to ensure that documents are delivered to their designated mobile phones and are handled only by their target applications only in their intended manners.

Porscha policies are enforced in two phases: the protection of content as it is delivered to the phone (in transit, see Section 5.4.2), and the regulation of content use on the phone (on platform, see Section 5.4.3). For the former, Porscha binds policy and ensures content confidentiality “on the wire” using constructions and infrastructure built on Identity-Based Encryption (IBE) [179]. For the latter, Porscha augments Android’s permissions-based security model to enforce policies by proxying content channels (e.g., POP3, IMAP, Active Sync) and placing reference monitor infrastructure within Android’s Binder IPC framework for mediating access based on content policy.

In developing such policy, we explore a broad array of applications obtained from the Android Market to understand the value and concerns associated with the documents they consume. We then describe our security requirements engineering based
content policy development methodology and present some resulting example policies. We implement Porscha on a T-Mobile G1 phone.

Later in the chapter, we evaluate the overheads associated with content policies and explore the volume and substance of security policies as represented by applications currently available in the Android Market. Our experiments demonstrate that Porscha is expressive enough to articulate needed DRM policies and that their enforcement has limited impact on performance. The delivery delay for MMS is slightly over 1 second, while latency from processing emails is only about 1 second or less without IBE and around 2 seconds with IBE. A security analysis is given and we conclude with a discussion of alternate designs and policy as well as infrastructure extensions.

5.1 Recognizing Risks to Content and Need for DRM

The evolution of the mobile ecosystem substantially accelerates data usage which has already become more prevailing than voice usage. These data-centric activities performed on the phones usually involve communication with external parties. Such activities are handled not only by the phones and the applications that come with them but also other custom applications obtained from mobile application market places or other sources. User needs for data communication is highly diverse. From financial transactions and buying airline tickets, to personal applications such as diaries and journals, the breadth of data processed by smartphones is immense. As a result, there is a strong need to be able to protect this content; for example, financial information should not be accessible through social networking applications or games. MP3-based MMS or photo content placed on Android smart phones should not be extracted and shared with impunity.
However, smartphones have very limited capability to deal with these protections on content. For instance, within the Android mobile operating system, there is no policy for enforcing how content from external sources is consumed and shared on the phone. As presented in Section 2.2.3, Android uses permissions to protect the phone platform from malicious applications and to control access to application components, but the policy is static and defined on the phone, rather than by external content providers. The result is that enforcing policy on specific information, such as an individual SMS messages, is impossible. As shown in Figure 5.1, there are no controls on information when it arrives at the phone. From some types of data such as SMS and MMS, any applications possessing proper permissions to receive those types of data are free to receive any data of such types (shown as solid lines in Figure 5.1). They can also pass the data to other applications using available inter-application interaction mechanisms (shown as dotted lines in the Figure). As increasingly sensitive information is handled by phone applications, the implications for ensuring the safe arrival and handling of information is enormous. This is a challenge for the current generation of phones: how can content produced by a provider be protected during delivery to the phone and on the phone itself?

To combat such mobile content handling issues, a consortium of mobile phone manufacturers including Nokia, LG, Motorola, Samsung, and Ericsson have recently developed standards for content protection on mobile devices. Codified within the Open Mobile Alliance and focusing primarily on pay-per-use content such as ringtones and multi-media, the OMA DRM v1.0 [180] and v2.0 [181] standards define an API and infrastructure for authorizing devices to process content. To simplify, OMA DRM devices obtain rights objects (use licenses and cryptographic keys) from providers that allow them to access downloaded content. The licenses can regulate how the content may be used in simple ways such as by discrete lifetime and maximum number of uses. The granularity of the OMA DRM specifications, however, is coarse. The licensing unit is the phone; as a result, the specifications say nothing about content management when it is on the phone. Specifically, there are no considerations of which applications may access content. This was reasonable when the specification was written in 2004, as there were no application markets at the time and phone manufacturers provided their own software for the phone. Since that time, though, the smartphone revolution has mandated a need for protections at the application layer, now that many applications can be purchased or downloaded to access on-phone content. Otherwise, content is subject to improper use by untrusted applications such as rogue media players.
Table 5.1. Number of sample applications that access SMS, MMS, and email.

<table>
<thead>
<tr>
<th>Application Category</th>
<th>Receive SMS</th>
<th>Receive MMS</th>
<th>Read SMS</th>
<th>Write SMS</th>
<th>Read Attachment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication</td>
<td>7</td>
<td>2</td>
<td>10</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Tools</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Finance</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Travel</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Others</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

In this part of the dissertation, we augment the Android permission model with a security infrastructure to enforce sender content policies. We place reference monitor hooks within the Android middleware at the point where an application could access potentially sensitive data. Our Porscha framework enforces fine-grained content policies over content delivered to the phone. It enables providers to ensure documents are delivered to their designated targets, and that the documents will be retrieved and accessed only by designated applications in compliance with specified policy.

5.1.1 Understanding Content Usage Through Applications Survey

In looking at DRM in smartphones, we must ask the obvious question of what must the service actually do. To answer this question, we surveyed applications and usage seen in current cell phones to attempt to ascertain what kinds of documents are commonly exchanged and what the reasonable requirements are that the providers may place on them. We evaluated the top 50 free Android applications in each of the 16 application categories present Android Market in April 2010. For the purposes of this initial study, we focused on applications that delivered content via SMS, MMS, or email. Table 5.1 shows the number of applications requesting permissions to receive SMS and MMS and to read from or write to SMS, MMS, and email attachments. We briefly summarize our findings on content use below:

Personal and Business Documents: Applications in the Communication category (e.g. third-party email/SMS/MMS clients), Tools category (e.g. anti-virus, backup tools, and Office Viewer), and Travel category (e.g. language translator) in Table 5.1 frequently manage personal or business SMS, MMS, and emails.

Documents in this category include sensitive emails between business partners and others encompass security capabilities, e.g., SMS is used for authorization in access.

\footnote{Throughout we use the terms DRM and content policies interchangeably. While the latter term is arguably more general, any distinctions are outside the scope of the definition and enforcement of policy studied here.}
control systems such as Grey [182]. In these cases, unintended exposure can leak business secrets or compromise the access control system. Thus, providers need to ensure that (a.) only targeted phones (i.e. authorized users) receive the documents, (b.) only trusted client applications can handle them, and that (c.) these documents can never be modified.

**Service-specific data:** A number of applications use SMS to send commands to on-phone clients. Note that in almost all cases, there should only be one legitimate client consumer for the content type—the one provided by the service itself. For instance, one spy camera application used SMS to command the phone capture pictures and record videos. Similarly, Mydroid\(^2\) is a tool for finding the phone by turning off the silent mode and turning up the volume when it receives a command via SMS, and Mobile Defense\(^3\) allows remote connection to the phones after receiving an authenticated SMS message.

Commands in these applications are sent or received via provided websites or other interfaces. Unauthorized exposure of the “command” documents could reveal the application behavior, and indirectly the user’s intent. The applications may misbehave if the commands are tampered with. In response, the senders must let (a.) only the phones under control receive the commands, (b.) only the applications to execute the commands to process them, and (c.) the commands read only and may be read only once, and (d.) ensure only legitimate content is consumed by the client.

**Financial Information:** Emails and SMS have become key media for financial institutions to communicate with their clients. For instance, banks and credit card companies offer SMS banking, SMS account alerts, and e-statements. Payment service providers such as PayPal and Amazon Payments mainly contact their customers via email and also offer SMS-based payment service. Similar to personal and business documents, the sending institutions aim to inform the users. As a result, the documents must be (a.) sent only to the phones of such particular customers. They must be (b.) accessed only by trusted messaging clients. Moreover, some documents such as payment requests may be designed to work with a group of payment applications trusted by the institutions which can also be identified by their hashes or signatures. In most cases, the senders should also ensure that (c.) these documents are read-only. They should be deleted only through trusted client applications.

---

\(^2\)http://code.google.com/p/mydroid/
\(^3\)https://www.mobiledefense.com/
5.1.2 DRM Policy Requirements

The surprising result of the application analysis was the incredible consistency of the content policy requirements. With few exceptions, the requirements fell into three categories:

- **Binding content to the phone** – most of the applications required that the content be targeted to a single identified user or phone. Failure to implement this policy could have catastrophic consequences (in financial applications), or undermine the entire service (in media access applications).

- **Binding content to endorsed applications** – often observed in desktop environments but largely ignored in the mobile device industry, it is important to control which particular applications can process protected content. The consequences of the failure to enforce this policy are similar to those above—a malicious application on an otherwise legitimate phone could corrupt, exfiltrate, or otherwise misuse delivered content.

- **Constraining continuing use of the content** – it is essential that the provider be able to control not just access, but how that access evolves or expires over time. Frequency, count, or temporal constraints were common. Failure to provide these polices would marginalize the license structures upon which many services are now built.

5.2 Understanding Content Delivery

The knowledge of the mobile phone environment in which documents including SMS, MMS, and emails are sent from external content senders to mobile phones is necessary for developing mobile content security solution. As presented in Figure 5.2, the delivery process is divided into two portions; first, the documents (referred to as documents in transit) are sent through the network to the phones, and second, the mechanisms on the phones deliver the documents (referred to as documents on phone) to the applications or allow the applications to access them. This section discusses background and limitations of the current security mechanisms in each of the two parts.
5.2.1 Documents in Transit

Figure 5.2(i.) provides an overview of how content is delivered to a phone from outside sources. For clarity, we refer to these external providers of content as content sources. Documents sent through the cellular (SS7) network, including SMS and MMS messages, are received at the phone’s Radio Interface Layer (RIL), processed by the baseband processor, and made accessible to the Phone application. Applications whose content source originates from the Internet, however, connect directly to them in order to receive these documents, such as email. There are no intermediaries on the phone to process this content prior to its handling by the application.

Lack of end-to-end security is a major problem in SMS, MMS, and email transport. SSL/TLS for email delivery only secures connection between the phones and the mail servers. SMS and MMS documents delivered through SS7 make use of security mechanisms found within the cellular network. The heart of the SMS system is the Short Message Service Center (SMSC) which receives short messages from mobile devices inside the cellular network or from external short message entities (e.g. web-based SMS portals). The messages are processed, stored in the SMSC queue, and delivered to the destination devices through a control channel. Messages in transit are encrypted by the network providers.

The MMS system centers on the Multimedia Message Service Center (MMSC). However, phones do not communicate with the MMSC directly but through a WAP gateway push proxy. Upon the arrival of MMS messages, the MMSC notifies the receiving clients with WAP push notifications over SMS. In response, the clients create a TCP/IP connection to retrieve the messages from the MMSC through the WAP gateway push proxy.
Clearly, the security of the WAP push notification delivery is based on SMS security. The accompanying MMS message retrieval can be secured using SSL/TLS. More information about the structure of the cellular network and its security is available in Section 2.3.

While SMS and MMS notifications are encrypted, there are still several security issues. GSM encryption is provided only over the radio interface since it is assumed that the SS7 network is inaccessible to external entities. The network providers do not always encrypt SMS messages [82]. Even if they did, though, the employed A5-family encryption algorithms have been compromised: a full rainbow table for the A5/1 cipher has been published [183], while an attack against the KASUMI cipher that is the basis for the new A5/3 cryptosystem can be performed in less than two hours on a PC [184]. Most importantly, GSM encryption does not give end-to-end security because the encryption key is shared between the mobile devices and the network providers, not directly between the content source and receiving device.

5.2.2 On-Platform Document Access

Access to documents that arrive on the phone is contingent on their method of delivery. Generally, it is controlled by both the phone operating systems, in our case Android, and the applications already possessing the documents. Figure 5.2(ii.) demonstrates how various document types are handled as they arrive at the platform. There are three cases that we consider:

1. **Initial Document Recipients**: These applications either receive documents (i.e. SMS, MMS, and emails) directly from the middleware platform or from system applications. Their access will be dependent on permission labels set within their manifest files as described in Section 2.2.1.

   To receive SMS and MMS documents, the applications must hold RECEIVE_SMS and RECEIVE_MMS permissions respectively. For both SMS and MMS, the applications require READ_SMS to read from and WRITE_SMS to write to the phone’s SMS/MMS repository. Lastly, to read the attachments of the emails received by the system’s email application, the READ_ATTACHMENT permission is needed. However, the Android’s static permission model treats all documents as equal. *All the applications with permissions to access a particular type of documents can access all documents of that type.* For example, an application with RECEIVE_SMS permission can receive any SMS. More importantly, these permissions are granted by the users who may not make good security decisions.
2. **Documents at Rest:** Some documents such as SMS, MMS, and the attachments of the emails received via Android Mail are stored in the Content Providers inside of the middleware, which act as databases for this content. Access to these Content Providers to either read or write data is also contingent on permissions in the application’s manifest file.

Similar to the case of initial document recipients, all the applications with **READ_SMS** and **READ_ATTACHMENT** can read all the stored SMS/MMS and email attachments accordingly. The applications holding **WRITE_SMS** can modify and delete any stored messages.

3. **Document Sharing:** *Indirect receivers* are applications receiving documents from other applications. In Android, the APIs allowing interaction and data sharing between applications are mediated by the **Binder** IPC mechanism. As described in Section 2.2.2, permissions are placed on application components to limit the data that can be sent and received between applications. This acts as a weak method of enforcing information flow enforcement, but is not secure as there is no concept of partial ordering with permission labels such that a lattice may be derived [185].

IPC allows the documents initially received by an application to be passed to other applications in the following ways:

- **Intent Messaging** - Referred to Section 2.2.2, Intent messages are used in most interactions including starting Activities, starting and binding to Services, and sending message to Broadcast Receivers.

- **Inter-application API Calls** - Using API calls (e.g. Service calls), applications can bind to components in different applications. For example, they can invoke methods in the binded Services and receive the returned results. In contrast to Intent messaging which the sending applications actively pass the content, the content returned as a result of such bindings is passive.

- **Content Sharing** - An application can share its data to other applications using Content Provider components as discussed above.

Android’s permission-based security model provides access control based solely on the possession of the permissions. It does not recognize the content being passed between components and thus does not support content security policy enforcement.
5.3 Threat and Trust Model

The threats against the documents are examined based on how they are transmitted to the phones and accessed by the applications as described in Section 5.2.1 and 5.2.2. We assume that the network is untrusted: an adversary is capable of subverting any communications received from the network interface, regardless of whether the cellular network or the Internet were transited. For instance, attackers are capable of eavesdropping SMS/MMS [23, 22]. Email eavesdropping is also possible if the senders or the receivers use insecure connections to their mail servers. This is unknowable to the opposite parties. Moreover, the attackers can spoof SMS/MMS and email documents. Porscha aims to provide security independently from that available from the underlying network. In particular, Porscha leverages IBE technology for content encryption and authentication to provide end-to-end security. We assume that PKGs are trusted.

Our trusted computing base (TCB) comprises the underlying Linux operating system and the Android middleware itself, as well as system applications. In commercial environment, these components cannot be modified without rebuilding the entire platform which requires the platform developer’s key (e.g. Google’s).

From each external sender’s perspective, the phone environment outside the TCB is partially adversarial. Some applications are benign for his/her documents while others are potentially harmful. Depending on the permissions the users granted to them, these malicious applications could read, modify, or delete the documents without the sender’s will. We assume that the external senders are able to identify the characteristics of the applications that they trust or do not trust to handle their documents. These characteristics can be as specific as the hashes of the applications or the developer keys used to sign them or as general as their configurations. The applications that are considered trusted to handle the documents are assumed to be interested in honoring the security policies associated with such documents. It is also assumed that the application developers safely store their keys thus the signatures on the applications can be trusted.

To mitigate the above threats, the external senders need to control access to their documents, i.e. to be delivered only to the intended applications on the designated phones and are handled only in intended ways. Straightforward encryption requires pre-existing relationship between the external senders and all potential receiving applications which disagrees with service oriented application model in Android. Moreover, it neither hides the documents’ existence nor regulates how they should be handled. As a result, expressive security policies are required. However, the senders who understand the importance
of the documents and the appropriate security policies for them cannot enforce the policies due to lack of control of execution and knowledge about the subjects (i.e. applications) accessing the documents. On the other hand, phone platform which is capable of performing the access control enforcement does not understand the threats to the documents and their security requirements. Porscha is designed to bridge this gap.

5.4 Securing Mobile Content with Porscha

In response to the limitations of Android’s security model for document handling, we have developed the Porscha framework. Porscha enforces security policy in two phases: initially as the content is transmitted over telephony networks and the Internet (as documents in transit), and thereafter as it is processed and stored on the phone (as on-platform documents). We begin this section by describing Porscha’s supported policies in response to our DRM requirements. Then, we illustrate how Porscha secures documents in transit and discuss our modifications to the Android OS to mediate the incoming and shared documents such that we can ensure their security on the platform. The type of policy rules and constraints initially supported by Porscha is discussed in Section 5.5.

5.4.1 Supported Policy

Porscha extends the XML OMA policy [180, 186] to embrace the new DRM requirements identified in our application study, discussed in Section 5.1.2. All policies are mandatory. That is, any OMA client receiving a Porscha policy must implement the extensions or deny access. Detailed below, policy is encoded in XML as either a section in the text part of MMS or email attachment (see Section 5.4.3). Porscha supports constraints on Devices – OMA DRM 1.0 binds the content to devices that have acquired the proper license (i.e., right object). OMA DRM 2.0 also supports binding to specific devices identified by the users’ International Mobile Subscriber Identity (IMSI) or WAP Identify Module (WIM). Described below, we extend identity to the phone number of the device itself (as regulated by the cellular provider).

Constraints on Applications – Porscha extends OMA DRM to constrain applications consuming content. Here the senders can specify access be restricted to applications with a given code fingerprint (hash of the application image), that are signed with a

\[\text{For brevity, we omit our XML structure and example policies beyond that identified in Table 5.1. Note that we are able to encode all policies uncovered in the application study using OMA and the Porscha extensions.}\]
given developer key, or require that the application be configured with a given set of permissions (this last policy is similar to those found in the Saint system [173]).

**Constraints on Use** – Common policies codified in OMA DRM, such as validity period and number of uses, are supported. We extend these to support not only the regulation of simple accesses, but also differentiation of simple access from read, modify and delete rights.

### 5.4.2 Document-in-Transit

Porscha must secure content delivery over the untrusted networks—namely the confidentiality, integrity, and authenticity of SMS, MMS, and email must be preserved. Porscha uses identity-based encryption (IBE) [179] to ensure these properties. IBE enables the senders to construct the public keys of the recipients from known identities (phone numbers or email addresses), thus eliminating the need for *a priori* key distribution. We use IBE in a standard fashion for messaging systems [187].

We briefly review the structure and use of IBE. Identity-based Encryption systems form a subclass [179] of public key cryptosystems. As with all public key systems, participants (users or other entities) are assigned a public key (which is widely distributed) and a private key (which is kept secret by the participant). What differentiates IBE from other kinds of public key systems is the public key itself. An IBE public key is an arbitrary string such as an email address, name, social security number or any other value that is desirable for the target environments. Serving a similar role to a CA in traditional PKI systems, each IBE system contains a trusted private key generator (PKG). The PKG generates private keys using system wide secrets and provides them to the participants through a registration process. The PKG advertises public cryptosystem parameters for the IBE instance.\(^5\) Encryption using the public key is performed by inputting the message (data), public key string, and cryptosystem parameters into the IBE encryption algorithm. Decryption is performed by inputting the ciphertext and private key to the decryption algorithm. Again, as in normal public key systems, it is also possible to encrypt using the private key and decrypt using public key (e.g., as used in creating digital signatures) [190, 191].

We use the following notation below. We denote the private key generator $PKG$, system parameters $PP$, sender (content source) $S$, and receiver (phone) $R$. The identity

---

\(^5\)The participant registration process, the secure acquisition process, and other key management services (e.g., revocation) have been discussed at length in other works [188, 189, 187]. Such issues are outside the scope of this work.
for participant $a$ is denoted $I_a$, the public/private key pair for $a$ is $K^+_a$ and $K^-_a$ respectively, content is $m$, and a policy for $m$ is $p_m$. Encryption and signature operations are denoted $E(d,k)$ and $Sign(d,k)$, where $d$ is the input data and $k$ is the key. We denote the one-time time ephemeral key used in the delivery of email as $k_e$.

**Securing SMS/MMS-in-Transit**

SMS/MMS delivery is performed by mobile network operators. In SMS/MMS delivery, the recipient’s telephone number (MSISDN) is used as their public key identity. A potential setup is shown in Figure 5.3. In this setting, each cellular provider runs a trusted Private Key Generator (PKG) responsible for its subscribers. The PKG server publishes the IBE public parameters via publicly verifiable medium, e.g., in a Verisign certificate. The phone’s private key and the system IBE parameters are loaded into the user’s SIM card by the cellular provider at subscription time. If the private keys can expire as discussed in Section 5.7, the cellular provider’s PKG authenticates the mobile
phone in the same way as when the mobile phone joins the network and issues a new private key to it.

During the one-time setup phase, the sender $S$ obtains the system parameters from the receiver’s $PKG_R$ to be used with the receiver’s MSISDN for encryption. If both the sender and the receiver share the same network provider, contacting each other’s PKG is straightforward as they also share the same PKG. In contrast, if they are in different networks, they contact each other’s PKG through their network providers using a multi-domain key management scheme such as those proposed for mobile phone environment [192, 193, 194]. The senders from the Internet (e.g. web portals) can contact the receiver’s network PKG directly. This step can be described as:

$$PKG_R \rightarrow S : PP_{PKG_R}, \text{Sign}(PP_{PKG_R}, K_{PKG_R}^-)$$

To send SMS/MMS, the sender encrypts its ID, the content, and policy using the receiver’s public key which is generated from the $PP_{PKG_R}$ obtained during the setup and the receivers MSISDN. The sender then signs the resulting ciphertext his/her own private key $K_S^-$. More precisely:

$$S \rightarrow R : E(\{I_S|m\||p_m\}, K_R^+) || \text{Sign}(E(\{I_S|m\||p_m\}, K_R^+), K_S^-)$$

Note that the addition of the policy, padding, and signature can increase the size of an SMS message beyond that supported by current networks (160 characters). Thus, as described in the following section, we convert SMS messages into MMS messages\(^6\).

When the MMS document is received by the phone, Porscha accesses the private key and transparently decrypts the incoming message, along validating the signature if the incoming message is signed. The signature validation process also requires the system parameters from the sender’s $PKG_S$ to be used with the sender’s identity.

**Securing Email-in-Transit**

Email delivery mainly involves Internet domain. We use a recipient’s email address as the target identity. PKGs run in support of email domains publish the public key parameters through extended attributes on DNSsec [195] MX records as presented in Figure 5.4. When the email account is setup, the client contacts the PKG. It is authenticated with the help from the mail server. The private key corresponding to the email address is

---

\(^6\)Maximum MMS size is 300 KB for T-Mobile and 600 KB for AT&T.
passed to the client. One-time key $K_{PKG,R}$ which is negotiated from Diffie-Hellman key exchange is used to encrypt and protect the integrity of the private key.

Similar to the SMS/MMS delivery, one-time setup phase allows the sender to obtain the system parameters $PP$ from the receiver’s $PKG_R$, to be used with the receiver’s email address for encryption.

\[PKG_R \rightarrow S : PP_{PKG_R}, \text{Sign}(PP_{PKG_R}, K_{PKG_R}^-)\]

Emails can be vary in size and be arbitrarily large. As performing IBE encryption on large content potentially incurs high overhead, the sender encrypts the body of the email using the one-time 128-bit AES symmetric key $k_e$, which is in turn encrypted with the receiver’s IBE public key. This is a common practice [196] and is employed in traditional public key cryptosystem construct compatible with S/MIME. The ciphertext is then signed, and the entirety is sent to the receiver as a MIME encoded email, as follows:

\[S : C = E(\{I_S||m||pm\}, k_e)||E(\{k_e\}, K_{PKG}^-)\]

\[S \rightarrow R : C||\text{Sign}(C, K_S^-)\]

Similar to above, the receiver verifies the sender’s signature (which requires the system parameters of the sender’s $PKG_S$) and decrypts the received construct.

5.4.3 On-platform Document Security

When a document arrives at the designated phone, Porscha enforces the content security policy specified by the content source. Detailed in this section, the Porscha mediator enforces policy through a set of protocol proxies and authorization hooks within the Android middleware. It provides complete mediation for all incoming documents to the phone, and further mediates document access across every Android component.

Policies are encoded in XML and specify access rights such as reading, modifying, and deleting data contingent on the specified application used to perform these functions. Due to their limited size, including policies in SMS messages is impractical. Therefore, we send them as MMS. MMS messages do not natively support XML attachments; as a result, we append the XML-encoded policy to the message as part of text attachment. We modified the Android Messaging application to demonstrate how the attachment process on the sender side is implemented. The Porscha Mediator converts them to SMS when received by the phone to prevent applications expecting SMS messages from being
adversely affected. For email, policies are included as attachments.

The policy attachments on the arriving documents are recognized and processed by Porscha mediator. However, there are several ways that the documents are received and accessed by the applications after they arrive at the phones, each of which requires different policy enforcement technique as follows:

### 5.4.3.1 Policy Enforcement on Initial Recipients

**SMS/MMS** — Depicted in Figure 5.5(a), SMS and MMS PDUs (the base structure for data messaging services in cellular networks) initially arrive at the Phone application (M1 in Figure 5.5(a)). Under normal (non-Porscha) circumstances, the Android SMS Dispatcher delivers these messages to all applications that have registered for `WAP_PUSH_RECEIVED` Intents (which requires the `RECEIVE_MMS` permission). The Intent identifies the URI to the MMS content. Messaging application then downloads the MMS content and stores it in the MMS Provider inside of the middleware. The MMS Provider broadcasts another Intent when the content is stored.

To enforce policy, Porscha delays the initial `WAP_PUSH_RECEIVED` broadcast, but automatically triggers the Messaging application to download the content (M2). Once the content download completes, a second mediator hook parses the PDU (M3), extracts the policy from the content, and determines, in conjunction with the Activity Manager
Service (M4), which applications are allowed to receive the document. Applications compliant with the policy receive the subsequent notifications, while those not in compliance will not.

Note that the Intent notifying the arrival of SMS/MMS cannot be issued from within the Android middleware: the broadcasting entity must be an application that possesses BROADCAST_SMS permission for SMS and BROADCAST_WAP_PUSH for MMS. Additionally, applications must be signed by the platform key for these permissions to be granted. To address this problem, we implement the Porscha Proxy as a system application signed and built with the platform. It receives from the Porscha mediator the list of the applications allowed to receive the document, the document content, the document metadata, and whether the document should be dispatched as SMS or MMS (M5). If the document should be dispatched as SMS, the Porscha Proxy constructs an SMS PDU and broadcasts an SMS_RECEIVED Intent to authorized applications (M6). If the document is dispatched as MMS, the proxy broadcasts a WAP_PUSH_RECEIVED Intent, containing the URI accessible through the MMS Content Provider, to authorized applications.

**Email** – Email traffic is opaque to Android: email client applications use application level protocols such as POP [197], IMAP [198], or Active Sync [199] to communicate with remote mail servers. For this reason, Porscha must “shim” email traffic by creating transparent proxies. The email enforcement intercepts email traffic at the network level through an SSL socket (E1) within the Binder network stack. Messages are intercepted and interpreted within each proxy and policy enforced. We use the Apache Mime4j library [200] to parse the e-mail message streams in plain RFC-882 and MIME formats. For each email, the XML attachments are examined. If the attachment is recognized as content policy, the Porscha mediator coordinates with Activity Manager Service to enforce the policy (E2). The content may only be retrieved by an email client if it satisfies the policy (E3).

For usability, rather that filter email from applications that fail policy, we chose to mask its content. In such cases, Porscha removes all information from the email’s header and body, and replaces these fields with the string Hidden. We will extend policy schema in the future to allow the sender to provide “alternate text” that would instruct the user where to go to obtain an appropriate application or license for the received content in the event the accessing phone/application does not satisfy the policy.

With Porscha policy enforcement, only the trusted applications by complying with the policy can be the initial receivers of the documents. They are expected not to leak the documents content to other untrusted applications. The next two mechanisms facilitate
Figure 5.6. Porscha Mediator intercepts cross-application content passing

them in enforcing the policies.

5.4.3.2 Policy Enforcement on Documents at Rest

By default, Android stores the SMS, MMS, and email attachments with the system applications using Content Provider components. Applications with permissions to access (read or write) these Content Providers can access this content even if they are not the initial recipients. To allow external senders to control access to the documents delivered from them, we add an extra policy field to the structure of each Content Provider record. The Porscha mediator inserts the policy (if available) into this field, and when the Content Provider record is accessed the corresponding policy is checked, and access allowed or denied based on the compliance of the caller application with the policy.

5.4.3.3 Enforcement on Indirect Receivers

As discussed in Section 5.2.2, the current Android security model for application interactions does not govern how the content of the documents can be passed between applications. To enable document content protection, the Porscha mediator allows the applications honoring the policy to pass the external senders’ policy along with the content that they hand to other applications as presented in Figure 5.4.3.3. The Porscha mediator intercepts the cross-application communication to verify whether the content passing should be allowed as follows:

- **Intent** – The Porscha mediator acts as a reference monitor for Intents that pass protected content. To let Porscha enforce the policy, the sending application binds the policy with the Intent that encompasses the content. The mediator prevents applications not satisfying the policy from receiving the Intent.
• **Content Sharing** – Inclusion of a policy field into Content Provider records, as described above, allows the Porscha mediator to ensure that every access to stored content satisfies the attached policy. Access is mediated through the Content Resolver mediation hook.

• **Inter-application API calls** – When an application API is called by another application (e.g. Service call), we bind a policy to the input parameter or return value containing content delivered in an Android's parcel object. The mediator interprets the parcel, and enforces the policy. If the policy fails (on either the call or return), a security exception is thrown.

## 5.5 Developing Content Security Policy

In order to protect their documents, the external content providers deliver their security requirements on content, represented in the form of security policy to the phone platform. The policy will be enforced by the platform such that access to the corresponding documents is regulated. As a form of access control policy, content security policy describes the subjects and the actions that are permitted or prohibited on objects – in this case, the documents. Based on the potential use cases of mobile content given in Section 5.1.1, this section introduces a methodology for developing the security for the content, which also aids in determining the types of policy rules that should be supported by Porscha. Next, the initial set of rules for Porscha content security policy is presented. We end the section with some sample security policies which can be used to protect the content involved in several key use cases.

### 5.5.1 Content Policy Development Methodology

The understanding of the documents’ functions and their potential threats are necessary for developing the security policies for them. Our methodology assists such process through five main steps as shown in Figure 5.7.

**Step 1: Identify Document’s Functional Requirements**

Each document delivered to the phone serves one or more specific purposes as shown in our application survey in Section 5.1.1. For instance, personal documents provide the human users with reports, appointments, trivia, etc. Some of them may contain private data. Financial and business documents are part of financial/business transactions which
may expect some response from the users. Some of these documents are sensitive such as emails between business partners and SMS used for authorization in access control system such as Grey system [182]. Location-related documents notify the users of the current locations of the senders who may concern about their location privacy. Application-specific data contributes to the application’s operations.

**Step 2: Determine Security Goals and Threats**

We consider high-level security goals such as confidentiality, integrity, and availability. Based on the functional requirements of the documents, we develop the threat descriptions which outline the scenarios that violate the security goals. As an example, unintended exposure of business documents can leak business secrets, violating the confidentiality goal. Violating its integrity with unintended modification potentially harms the business. In the case of the Grey system, such exposure can compromise the access control system, while tempering with it may render the system inaccessible. Similarly, unauthorized exposure of application-specific documents reveals the applications’ operations. Moreover, the applications may misbehave if the commands are tampered with.

**Step 3: Identify Document’s Target**

Based on the functional requirements and potential threats, the external senders identify the intended recipients of the documents which are the actors in the security policies. As reflected in Section 5.1.1, the recipients can be the users and/or applications. In the first case, the intended recipient can be the default application for viewing such type of
content. In the later scenario, the external senders may know the specific set of target applications or the properties or the type of functions of the target applications. For example, most of the times, personal messages target only the human users. Payment request documents may also be designed to work with a group of payment applications trusted by the document senders. Location-related documents may be designed to be inputs to some location visualization or social network applications. Applications’ control command documents are to be consumed only by those particular applications.

**Step 4: Specify Security Policy**

Security policies for mobile documents constrain the operations that active entities, namely the applications on the phone, can perform on them. To protect the confidentiality, integrity, and availability of the documents, the policies restrict read, modify, and delete operations respectively. A conservative approach is to assume that the applications other than the target applications are untrusted and are potentially malicious. As a result, the security policies deny all operations except for those that serve the purpose of the documents identified in Step 1 and are performed by the targets specified in Step 3. A more flexible approach is to deny only the operations from the actors that potentially compromise the threats identified in Step 2.

For example, to avoid the threats to personal, business, and financial documents, the senders may create the policies that limit the target applications only to the set of trusted client applications and the set of operations allowed to be performed (e.g. disallow any modification). Moreover, some documents such as payment requests may be designed to work with a group of payment applications trusted by the institutions which can also be identified by their hashes or signatures. As the senders know the exact target applications of the application-specific documents, the policies may specify the applications by the hashes of the application executables.

**Step 5: Determine Security Mechanism Limitations**

Lastly, some security requirements may not be possible to fulfill within the limitation of the mobile phone platform. For instance, the email clients that satisfy the external providers’ security requirements may not be configured to connect to the mail server from which the emails arrive. Moreover, the information about the property and capability of the applications on the platform is limited. Therefore, the security requirements need to be revised to comply with the limitation.
5.5.2 Content Security Policy Rules

Based on our investigation on the types of documents delivered to the phones and their security requirements, we create an initial set of classes of policy rules supported by Porscha. A security policy is composed of a set of allowed actions together with a set of properties of the applications allowed to perform each of them and the context conditions under which the operation can occur. The current Porscha content security policy grammar is presented in Figure 5.8.

Each policy regulates read, modify, and delete operations. The set of allowed actors is defined as the applications that satisfy certain conditions for default deny policies and as those do not satisfy such conditions for default allow policies. Porscha currently recognizes four main types of application conditions as outlined in rule (5.8) in Figure 5.8. For strict policies, the applications can be identified by the hashes of the applications’ executables which uniquely identify them. In some cases, the external senders do not know the exact applications but know which developers can (or cannot) be trusted. Therefore, the set of allowed applications can be defined as those signed (or not signed) with the developer’s keys in the given set. The intended receiving applications can also be identified by their configurations such as the set of permissions they possess. Alternatively, the senders can identify the target application as the default application for a particular action. In addition to conditions on the actors themselves, the access to the documents can also be controlled by the phone’s context information such as location and time as shown in rule (5.11).

5.5.3 Sample Content Security Policy

Consider various types of documents presented in Section 5.1.1, Porscha policies can be used to secure them. Some examples are presented in Table 5.2. To protect personal or business email, the sender may create the policy that limit the target applications only to those from the trusted application developers specified by their signatures on the applications such as in policy 1. Trusted developers may include Google and the enterprise itself. The policies can also describe the phone context under which the operations can occur. Policy 2 states that the authorization SMS in our Grey access control system example can be read only if the phone is within 50 meters from the place under control. For application-specific documents, the senders usually know the exact target applications. As a result, the policies may specify the applications by the hashes of the application executables as presented in policy 3.
When the documents expose the senders’ locations, privacy concerning senders may restrict access to these documents only to trusted client applications. Moreover, location visualization applications may have read access to them to present the information to the users. They may be indirectly specified by their permissions to access the user’s location as shown in policy 4 or by the functionality they offer which is exposed by their registered Intent Filters described in policy 5.

5.6 Evaluating Porscha

This section briefly evaluates the costs of policy enforcement in Porscha. All experiments were executed on a HTC G1 Dream smartphone over T-Mobile 3G services. Porscha was built on the Cyanogen [29] Android 2.1 firmware build and installed on the phone. Our implementation uses Stanford IBE library V.0.7.2 which is a C implementation of the Boneh-Franklin identity-based encryption scheme [179]. Hess signature scheme is used to produce our ID-based signatures, implemented using the Pairing-Based Cryptography (PBC) library [201]. We parameterized the PBC library to use pairing constructed on
Table 5.2. Examples of security policies for content protection.

supersingular elliptic curves (SS) [202] for fast cryptographic pairing operations\(^7\). The IBE modules are crossed compiled for ARM architecture and stored as executables inside the platform.

The experiment is performed for document delivery with and without the policy controlling read access by restricting the accepted application executable’s hashes. All documents are received via 3G data connection. Each experiment was repeated 10 times and the average reported (with negligible observed sample variance).

Highlighted in Table 5.3, an initial set of experiments sought to measure the overheads associated with SMS processing. Here we measured the time between the arrival of the PDU and the time it is dispatched to consuming applications. The experiments showed that SMS processing time is less than 0.1 seconds in unmodified Android. SMS documents are delivered as MMS introducing an additional 4 or greater second overhead. Microbenchmarking of SMS processing revealed three central underlying costs: MMS push notification handling (\(\approx 1.03\) s), MMS content retrieval (\(\approx 1.44\) s), and other connection management processing (\(\approx 1.04\) s). The lower costs associated with SMS-over-MMS vs. MMS with media content were associated with the reduced size of the objects being downloaded (SMS policy objects were on the order of 100s of bytes versus 18kb .jpg objects in MMS experiments). The maximum observed overhead for IBE was about 480 msec.

For MMS, we measure the latency from the arrival of the PDU to the time to MMS content is completely downloaded and applications notified. Without IBE, Porscha

---

\(^7\)This is the default a-type pairing of the PBC library created on the \(Y^2 = X^3 + X\) curve over 512-bit base field, using a Solinas prime as group order. It provides 1024-bit DLOG security.
incurs about a 4% overhead ($\approx 20$ msec). IBE adds about 1.3 s. to the overhead—significantly more than in SMS. Here again the cause is the size of the encrypted media: the `.jpg` object. Note that recent advances in IBE offer run-time improvements that can reduce these overheads by as much as 20-35% [203], and techniques such as the use of one time symmetric keys (as detailed in section 5.4.2) may substantially reduce these costs.

The overhead of processing email without IBE ranges from 0.7 seconds to just over 1 second, depending on the email access protocols. These costs are largely due to the proxying of the access protocols, SSL, buffering, email message reconstruction, and policy extraction and evaluation (if policy is available). IBE incurs approximately 1.3 - 1.9 s, also depending on the protocols.

## 5.7 Discussion

This section examines the security guarantees provided by Porscha and potential threats to these guarantees.

### 5.7.1 Protecting the Private Key

Porscha’s model for key distribution involves the client phone receiving a private key from the phone provider. Recall from Section 5.4.2 that a private key generator (PKG) is trusted to create IBE keys and that a phone’s MSISDN can be used as a public key identity. The cellular provider will operate the PKG and provide the private key at subscription time on the client’s SIM card. However, the SIM is merely a memory card, and is susceptible to being stolen or lost. Therefore, we use a shared secret between the provider and client phone to encrypt the private key, with knowledge of the secret required to unlock the key package. One way of communicating this to the user would
be with on a slip of paper or a similar out-of-band method when the SIM is purchased or reprogrammed. This method is already used for the SIM authentication key, stored by the provider in their authentication center (AUC).

5.7.2 Recipients Without Porscha

External senders do not have prior knowledge about whether Porscha is available on client phones. In the absence of Porscha, the clients would not be able to access protected documents. This is reasonable, as any content delivered to and intended for the phone should remain opaque to the user. As a result, whether it be phones that do not have a Porscha framework installed or another means of accessing content, e.g., retrieving emails on a computer, content protected by Porscha should and will be inaccessible by these entities.

Note that emails accessed by the IMAP protocol are ultimately managed by an IMAP server; thus, any modifications made to an email by Porscha may be reflected on other clients. To resolve this issue, we store all modifications, such as decrypted emails and those with information removed, locally on the phone, and only reflect back to the IMAP server the original email that was sent to the phone - thus, an original copy of the email is always maintained.

5.7.3 Application and Platform Trust

With Porscha, we are making assumptions of trust in the Android middleware and associated system applications. There are a number of reasons why this level of trust can be considered appropriate. First, Android applications are signed with a certificate whose private key is held by the system developer; this provides a means of ensuring that the application’s integrity is intact and that the origin of the code is as expected. Tools such as Kirin [204] allow install-time certification of applications against potentially dangerous functionality.

Android is middleware that runs on a Linux kernel. Several methods of ensuring kernel integrity have been considered, and this is an area of ongoing research. These include run-time monitors such as the Linux kernel integrity monitor (LKIM) [205]. Ensuring that the phone platform itself is booted into a trustworthy state has also been an area of considerable focus. One promising solution is to include trusted platform modules (TPMs) [206] inside mobile phones; specifically, the Mobile Trusted Module (MTM) initiative [207] has considered a TPM-like device that adds functionality for secure boot, which enforces integrity protection of the underlying firmware and system
state, and allows for continual assurance of boot-time guarantees through use of the Linux Integrity Measurement Architecture (IMA) [208].

5.7.4 Alternative Application Enforcement Infrastructures

The extent to which Porscha protects a document largely depends on the attached policies. The senders can indicate the target applications by different degrees of specification from unique application package hashes to loosely defined application properties.

An even stronger security model can be implemented as a performance trade-off. For example, Porscha can be used along with Saint [173] which regulates application interactions (but does not examine the content being passed). As a result, we would gain a more comprehensive view of application behavior and could ensure that all applications are monitored for sharing violations.

Adoption of more heavyweight mechanisms offering continual content monitoring, such as dynamic taint analysis [209, 210], is also possible. However, these systems are not designed for information with semantically rich policy attached. The policies for incoming documents are mostly unique. Managing large and highly dynamic set of taint markings (e.g. in taint analysis) can thus be burdensome. Porscha’s content enforcement mechanism is comparatively quite lightweight.

5.7.5 Digital Rights Management

DRM has been contentiously discussed for nearly 15 years. Such controversy stems from the primary application of DRM: to restrict the use of digital content and prevent piracy, ostensibly to preserve artistic integrity and protect revenue streams for content creators. For example, three competing DRM technologies for mobile or portable devices are Apple’s FairPlay [211], Microsoft PlayReady [212]. Along with OMA DRM, all aim to limit media usage for commercial purposes. In Android, OMA DRM 1.0 is supported to manage ringtones, MMS, and pictures, preventing users for forwarding these documents. An external generic framework for DRM implementation is also available but is not used by the official platform. An external framework containing the OpenCore Content Policy Manager (CPM) [213] is also available. CPM does not implement any DRM algorithms or protocols, but acts as an aggregator with interfaces for authentication, authorization, and access control. Plugins are available for multiple DRM agents, such as WMDRM [214] and DivX [215].

There has been significant opposition to DRM [216, 217] with detractors viewing it as a means for limiting consumer rights and eliminating “fair use” provisions. However,
DRM is by definition a generic term for access control technology that secures content and limits its distribution [218, 219]. We consider DRM’s role in Porscha strictly as a means to providing content-based access control without comment on business, legal, or philosophical issues. We differentiate from existing DRM schemes, however, and provide a superset of functionality by preserving confidentiality, integrity, and availability, not merely employing encryption and licensing as with typical DRM implementations. In addition, Porscha is lightweight and designed with mobile solutions in mind; by contrast, many advanced DRM protocols are heavyweight and not transparent to applications.
In the era of mobility explosion, viewing mobile telecommunication environment as mobile ecosystem allows us to understand the challenges encountered in this modern paradigm including security concerns. In this opening and converging environment where multiple stakeholders and their operational components increasingly coordinate, the stakeholders need to protect themselves. However, achieving such security is non-trivial especially when they are not initially designed to support such rich and highly integrated collaborative model.

Throughout this dissertation, we demonstrate a systematic avenue for securing mobile phones. In particular, we unveiled two main difficulties, namely the absence of a proper definition of secure phones, and the lack of infrastructure to impose such definition on the phones. Furthermore, by considering phone-centric model, we identified three main elements that influence the fidelity of mobile phones, namely the mobile phone platform which is the fundamental of all operations, the mobile phone applications which are directly responsible for transactions and data, and mobile content which are consumed by phones and applications. Accordingly, we seek to define and enforce secure phones from these three perspectives.

Secure phones can be defined as those behave within the expectations, namely security requirements, of the stakeholders. Correspondingly, we explored the benefits of security requirements engineering in obtaining such requirements, formally articulated as security policies. More specifically, we proposed a three-prong methodology for developing appropriate security policies to secure mobile phone platforms, mobile phone...
applications, and mobile content, reflecting the requirements of users and platform developers, application developers, and content and service providers with respectively. For each of these three facets, the requirement space was explored and potential set of security policies was created based on real world scenarios. Finally, we provided pertinent security infrastructures that accept security policies from such stakeholders and enforce them on the phones.

It should be emphasized that while our research focus is in mobile phone landscape as our goal is to secure mobile phones, our strategy to approach security is general enough to be applied to other segments of the mobile ecosystem. In addition to mobile phone platforms, mobile phone applications, and mobile content, the technique can also be expanded to support other perspectives which have not been addressed in this dissertation including cellular networks.

This final chapter analyzes some important issues uncovered during the course of our investigation together with the lessons learned. We also describe several key factors that impact our design decisions. Lastly, we highlight the needs for further investigation into security issues particularly as new subsystems are progressively introduced to the mobile ecosystem and conclude this dissertation with a discussion on a potential future research agenda.

6.1 Security Policy Framework

An important characteristic of the mobile ecosystem is extensive coordination among stakeholders. Unfortunately, this highly collaborative model introduces complex security requirements, which in turn drive the need for more complex policy framework to accommodate them.

Our scenarios in Chapter 3 and 4 are great representative cases of this reality. It was clear to us that a fundamental problem in Android security model comes from the limitations of its security policy framework, particularly, policy expressibility. From the stakeholders’ perspective, highly expressive policy allows them to describe their security requirements more forthrightly. Unfortunately, the permission label-based policy by itself is barely capable of conveying security requirements. Label-based policy by nature is inexpressive. It contains low operational semantics. In other words, the enforcement of this class of policies is described less directly and requires translation. Namely, it cannot give comprehensive and straightforward description about the operations being controlled and the conditions under which they should be executed in the forms that the
enforcement mechanism can easily follow.

Even outside of security domain, such semantic problem is a main challenge in building systems for multi-stakeholder environment. It is crucial to establish a common understanding of the security policies between the stakeholders. However, label-based security model has historically been used to provide access control in single security domain such as in lattice-based access control [124] and label-based access control [220] systems, where security labels are created and managed by the system’s security administrator. When applied to multi-stakeholder system, Chapter 4 demonstrated that this label-based security model is not practical for such loosely coupled environment. This stems from the fact that the security policy by itself lacks in semantics. The labels need to be further translated. Nevertheless, when security labels (permission labels in case of Android) are created by different stakeholders with no pre-existing relationship, ensuring consistent meaning of each label across the system becomes troublesome.

Based on such observation, we opt to use security policy framework that employs more expressive policy. The actors, operations, and conditions are explicitly specified. However, our policy framework is not the only possible solution. It is witnessed that label-based policy framework is successfully adopted in other distributed systems such as JPmail [221]. Nevertheless, these systems provide semantics management systems to enable the involved stakeholders to develop a common mapping between the labels and their semantics. In contrast, Android does not provide any additional support or out-of-band collaboration, the stakeholders cannot interpret the labels created by other stakeholders.

Therefore, in designing security policy framework for multi-stakeholder environment, the semantics of the policy is a prime concern. A preferable practice is to use expressive policy which contains rich semantics. However, if less meaningful policy such as label-based policy is adopted, it is important that the system developers are aware of this critical issue and provide well-articulated semantic infrastructures.

6.2 Retrofitting Security through Security Requirements Engineering

Retrofitting security has long been widely criticized in security community. As shown all through this dissertation, our attempt to incorporate further security into existing phone system built with insufficient protection has been traumatic. Even so, we could only partially address the challenges. Since the system has not been initially designed
to support such rich security model, some security requirements cannot be satisfied due to some inherent limitations which cannot be removed without modifying most part of the system or totally reconstructing it.

From our experience with security requirements engineering, we learned two main lessons in adopting this principle. First, while SRE is very beneficial, the process is considerably labor-intensive. Particularly, retrofitting security requires a comprehensive understanding of the system as well as the existing security model. Based on our developed methodology, we observe that collecting the input to the process can potentially be automated. It is also feasible to develop automated supporting tools to facilitate and guide the SRE process [222]. However, by the nature of the task, some steps require human expertise, experience, intuition, and creativity thus are impractical to automate. Second, we learned that even though we could devise the stakeholders’ security requirements, it is still unclear how these requirements can be evaluated. While several approaches for evaluating the system’s security or its implementation against some predefined security requirements or criteria are available, we have not witnessed any standard metrics or concrete proposals for evaluating whether the developed security requirements are the correct ones in security literature.

It can also be observed that the security requirements engineering methodology proposed in this dissertation is tailored to Android. Nonetheless, at high level, our methodology is general enough to apply to other mobile phone platforms because they operate under the same mobile ecosystem and face almost the same challenges. A possible approach to provide such generalization is to separate the general part of our methodology from the Android-specific part and create a generic platform-independent module. The platform-specific portion can be then created for each platform as a pluggable module to the generic one.

### 6.3 Trust Model

Every security system must be built over a trust model. Throughout this dissertation, we assume trust computing base (TCB) consisted of the Android middleware platform and system applications. We believe that this is a reasonable assumption because in commercial settings, the middleware cannot be modified after the phones are released. In Chapter 5, one key additional assumption is that the applications considered trusted by the external content providers would honor the security policies associated with the documents they received.
However, there is no notion of correct trust model because all individuals have their own ways of justifying trust even if they take the same role in the environment. It is important to stress that by shifting the trust model, the security model will need to be adjusted accordingly such as those discussed in Section 5.7. It is still almost impossible in the current state of the art to build security system that is flexible enough to accommodate every customized trust model.

6.4 Future Work

Mobile phone and telecommunication network security has progressively gained research attention over the past few years largely due to the proliferation of smartphones and the increased number of attacks against them. However, the dynamic of mobile phone and cellular technology together with the recent evolution of the mobile ecosystem has opened even more research opportunities.

In addition to mobile phone industry, many other business domains have also appreciated the power of smartphone platforms which can be much more than operating systems for smartphones. For example, Android have been ported to commercial portable computers [223, 224] and planned to be adopted to set-top boxes [225]. Smartphone applications working hand-in-hand with in-car system are now available [226]. Moreover, Android-integrated automobile has also been invented [227]. As this technology is increasingly pervading other aspects of our life, there will be further unexplored research areas beyond mobile phone security.

This dissertation is only a small representation of the required contribution to the journey to secure mobile phones and secure mobile ecosystem. There are still a lot of remaining issues such as in securing the network and commercial adoption of TPM to mobile phones. There would also be more unforeseeable challenges particularly when more functions are introduced such as the deployment of telecom web services, more stakeholders become part of the ecosystem, and more technological domains are connected as an effect from industry convergence. In this section, we give a high-level overview of several potential next steps in this research area.

6.4.1 Confining Applications with AppJailing

Our Saint framework presented in Chapter 4 and Porscha framework described in Chapter 5 restrict the operations of the mobile phone applications such as initiating interactions and receiving content by placing authorization hooks to mediate predefined API
calls. However, this technique requires thorough middleware source code inspection to ensure complete mediation. The achieved level of restriction is also limited by the capability of the defined access control policy and enforcement mechanisms. More importantly, it does not regulate the operations performed without using the available APIs such as side channels.

In response to such limitations, we seek for a more rigorous and completed approach for constraining Android applications to restrict application interactions and information flow, which will also prevent side channel attacks. Moreover, one of our main design objectives is to have a security infrastructure that is transparent to the applications. From the applications’ perspectives, they are running on a fully functional platform. Normal application interactions must be allowed as long as such interactions are considered secure. Given these requirements, a promising solution is to jail groups of applications and seamlessly limit their operations within the jails and their communication with other entities outside of the jails, based on their level of trust and functional requirements.

From our initial survey of existing approaches, there are a considerable number of proposals for application/process containment. Security literature refers to general techniques for confining programs or processes as sandboxing, which usually aim to enable secure execution of untrusted programs and software fault isolation. It can be achieved through two main approaches, namely access control and isolation. Sandboxing using access control allows the sandboxed processes to see all the resources. However, access control layer is inserted to regulate the access to them. The approach normally restricts the processes’ functionality for security. System call interception-based approaches generally involve intercepting system calls before they are executed. This allows user to deny, audit, pre-process, post-process, and replace existing system calls. The sandboxing mechanism may reside in the user space [228, 229, 230] or the kernel [231, 232]. Processes developed in some particular languages are sandboxed by their runtime environment such as Java Virtual Machine [233] and Safe-Tcl interpreter [234] which provide security check to limit the execution of untrusted code. It can be observed that access control-based sandboxing approaches contain each process by limiting its capability. More importantly, they are designed to jail individual applications not groups of applications.

The second approach to sandboxing is through isolation. Unlike access control-based approach, isolation techniques allow the contained processes to see only subset of resources. The most prevalent solution is isolation using virtual machines. There are two main types of VMs. First, host-visualization has host OS running atop the hardware
and guest OSes running over the host OS. Third-party virtualization software such as VMWare [235] and VirtualBox [236] is used to create and manage the guest OSes. Each guest does not aware of other guests. The second type is paravisualization. It has a small, low-level hypervisor such as Xen [237] which is a virtualization software that runs an operating system on the system hardware. It is the first control software loaded when the computer starts, and has one or more privileged guest OS to handle the actual device drivers. Both of theses are in domain 0. Then, there are other OS. Normally, OS must be modified (ported) to call Xen virtual drivers which in turn call the real drivers.

Shared kernel or operating system-level virtualization provides kernel-level sandboxing. Also known as a chroot jail or container-based virtualization, this technique utilizes the chroot feature to change the root file system of a process to provide isolation. The chrooted system is created such that it believes that it is a standalone system with its own root file system. A good example is Solaris zones [238]. A zone is an OS abstraction that acts as an isolated virtual server. There is one global zone which hosts non-global zones. Two main types of non-global zones are available. Small zones share most of the OS with the global zone; thus, can be viewed as a software partition. On the other hand, big zones have their own operating systems included; thus are akin to virtual instances of the operating system. Each zone provides a protected environment to protect applications from one another for software fault isolation. Since it is surrounded by security boundary, processes associated with a zone cannot interact with or observe processes in other zones.

Isolation-based jailing especially zoning allows application grouping which conforms to our design objective. Small zoning also allows such functionality without the need to visualize the entire operating system. Nevertheless, the available isolation techniques including zoning totally segregate applications, forbidding communication across zones. As a result, they are not suitable for our scenario.

We propose AppJail as a more flexible jailing system for Android which allows functionality including cross-application communication while transparently providing security. AppJail provides isolation according to security policy by grouping applications and jailing them based on their functions and their level of trust. We do not aim to enforce total isolation among jails but to control in-jail and cross-jail interactions based on security policies. In developing AppJail, we need to investigate various security policies such as jail assignment policy which controls how the applications are grouped and put in the jails, in-jail policy which regulates the behaviors of the applications in the jails, cross-jail policy which restricts the interactions of the applications in different jails, and
potentially jail hopping policy which allows the applications to move from one jail to another. To enforce these policies, we aim to implement AppJail confinement infrastructure in the Android middleware layer which allows jailing without the need to modify the kernel.

6.5 Cloud Computing Infrastructure for Securing Elastic Android Applications

Today’s smartphones are functionally competent to provide utmost application power which would accommodate feature-rich applications ranging from games, banking, and social networking to image processing and data analysis. However, this hypothetical capability is limited by their available resources. As an example, Google’s Nexus One phone [239] comes with 1 GHz CPU, 512 MB RAM, 4 GB Micro SD card, and 7.2 Mbps HSDPA 3G data connection. The battery allows up to 10 hours talk time on 2G or 5 hours Internet use on 3G. While this device is considered an upscale mobile phone, it is still incapable of running resource-intensive applications. A recent paradigm shift has introduced a novel computing model known as cloud computing which offers Internet-based computing. Cloud providers supply cloud infrastructures such as Amazon EC2 [240] and Microsoft Azure [241] which deliver resources as services. They enable their clients to avoid capital expenditure on IT infrastructure and IT experts. Moreover, this model provides resource-on-demand which allows the clients to seamlessly off-load some tasks from their local computing platforms including mobile devices to the cloud. By transparently leveraging cloud resources, it is possible to augment the capabilities of resource limited devices, such as mobile phones, including CPU, memory, storage, network bandwidth, and power.

Among several research attempts for distributing on-phone processes to the cloud, the existing on-platform security model has been disregarded. Such security protection should have been provided if all the tasks were executed on the phones. In particular, Android provides a reasonable amount of cross-component and platform security which should not be ignored. This part of our research plan aims to provide a framework that allows the application processes to be dispatched to the cloud while preserving the existing on-platform security.

An early work on migrating mobile phones’ processes to the cloud clones the entire mobile phone system image to create off-device runtime environment in the cloud [242]. The status of the phone is periodically synchronized to its replica in the cloud. While
this approach can offload the mobile phones without (or very minimal) modifications to
the applications, it brings in complexity if the tasks need to access physical hardware
on the phone. Moreover, it impedes parallel execution of the applications over different
cloud nodes [243].

A more recent approach introduced elastic application model, which enables
application-level augmentation [243]. In this model, applications are created to be cloud-
aware. They consist of small multiple components called weblets, which are autonomous
software entities that can migrate between the device and the cloud and expose REST-
ful web service interface to communicate with other weblets within the same applica-
tion. Cost-effective execution configuration regulates the weblet migration based on the
device’s status such as workload, battery level, memory usage, and also user prefer-
ences. In terms of security, the accompanying works [244, 245] proposed several security
mechanisms to enable cloud platform integrity measurement, authentication and secure
communication among weblets of the same application, and authorization of weblets for
accessing third-party services.

As Android applications are already component-oriented as described in Section 2.2.2,
bringing elastic application model to Android should be less troublesome. The compo-
nents would appropriately represent the units of execution akin to the weblets. How-
ever, by moving Android application components into the cloud, we are facing two main
challenges. First, a core design objective of Android is to enable cross-application in-
teractions. The deployment of such rich application interaction model has not been
addressed in the existing proposals. The second key challenge is the integration of the
existing on-platform Android security model presented in Section 2.2.3 and potentially
the enhancement we proposed throughout this dissertation into the model. The loosely-
coupled nature of the elastic application model would make component access control
more difficult.

To achieve secure elastic application model in Android, we plan to encapsulate the
on-cloud components with security check layer which completely mediates the inter-
component communication to enforce access control. Every weblet must carry the set of
security policy related to it. For performance, the execution configuration which includes
the migration preferences must also take the set of potential interactions as allowed by the
security policy into account. This part of the research would explore related engineering
challenges as well as the development of appropriate migration policies taking potential
interactions among weblets into account.
6.6 On-Platform DRM for Application Usage Control

Today, nearly 40% of the applications on the Android Market are proprietary applications [246]. A primary concern of the developers of these paid applications is to ensure that they receive proper payment from the application users. Furthermore, the application developers may wish to control how their applications are used on the phones. This would also create an alternative marketing model for mobile phone applications such as availability of applications for rent and applications with custom features.

DRM technology has been invented to restrict access to the content to only authenticated devices. As briefly discussed in Chapter 5, Android and many other mobile phone platforms provide very little support for DRM and is primarily used to restrict access to multimedia content. To provide DRM for applications, Android Market provides copyright protection for paid applications. The application developers can select this option when placing their products on the Android Market. Nevertheless, the available scheme only supports forward-locking which prevents the users from copying the applications across devices. Moreover, there was some evidence that the existing scheme renders some applications inoperable [247, 248] and does not provide a complete protection [249]. The solution such as Porscha proposed in Chapter 5 does not control application execution. The Kirin framework presented in Chapter 3 regulates application’s installation based on the user and platform developer’s policy not the application developers’ policies. More importantly, it does not control the execution of the applications during runtime.

This part of our research agenda seeks for a more flexible and complete DRM solution for Android applications. Our goal is not only to prevent application forwarding but also to regulate application usage. Thus, even though the users download the full version of the applications, the extent to which these applications run is regulated by the associated constraints which depend on the purchased application features. By separating the constraints from the applications, we avoid the need to explicitly implement the feature constraints into the applications or to create multiple versions (e.g. demo and full version). It is also enable flexible usage control configuration without modifying the applications. Moreover, the constraints can be changed even after the applications are installed, such as when more features are purchased or some add-on trials have expired.

OMA DRM [250, 251] is a prevalent standard for mobile phones. It was developed by Open Mobile Alliance to initially control copying of cell phone ring tones and has been extended to control access to media files, such as music files and videos. The OMA DRM standard provides specifications for content formats, protocols, and a rights expression language. Application developers are directly affected by the way the content
is distributed and consumed. According to the standard, a client component called DRM agent provides a mechanism to control not only piracy via the mobile network but also via local connections, such as Bluetooth, infrared, and Multi-Media Card (MMC) memory cards. The most basic form of protection is forward-locking which prevents content from leaving the mobile device. A higher level of protection can be achieved through the use of right objects which contain permissions and constraints to the DRM content. They are generated by the rights issuers who are normally part of the content providers. The right objects are linked to the media objects and are used to govern their usage. Each right object is protected by a rights encryption key (REK) which encrypts its sensitive parts such as the content encryption key (CEK). The CEK is used to encrypt the media as the means for access control. It also contains usage rights information that allows the user to play, display, execute, or print the DRM protected media object, and usage rights constraints that restrict the number of times or the period of time during which the media object can be used.

Our design goal is similar to OMA DRM in a way that we aim to regulate the products’ usage comparable to that is allowed in OMA DRM through the use of right objects. A potential design for building an appropriate DRM scheme for Android applications is to augment the Android platform to provide DRM policy enforcement controlling the applications’ runtime behaviors. In this design, the application developers include a usage policy as a configuration file into each of their DRM-protected applications. Furthermore, the platform must be attested by the application distributors such as the application market places to ensure that it provides forward-locking and proper policy enforcement. Lastly, we must explore the set of application usage policies in response to the application developers’ requirements while are enforceable by the platform.

6.7 Summary

The evolution of mobile ecosystem has not only manifested a large number of existing mobile phone vulnerabilities but also introduced countless new ones. This dissertation identified two key challenges in securing mobile phones, namely the lack of proper definition of secure phones and the insufficient security infrastructure to impose it on the phones. We explore the benefits of security requirements engineering in identifying the security policies which reflect the definition of secure phones from the viewpoint of different stakeholders. Accordingly, we provide pertinent security infrastructure to enforce such policies on the phones. We approach security from the perspective of the mobile
phone platform, the mobile phone applications, and the mobile content, which are three fundamental elements that influence the fidelity of the phones.

Mobile phone security is a vast research topic. This dissertation represents only a portion of the investigation in this landscape. There is still plenty of room for improvement and a myriad of research opportunities. As this evolution continues, there will be a lot more emerging products and services as well as new players joining this immense ecosystem. Undoubtedly, security will remain a critical concern. While engineering challenges could be different, the strategic approach to security learned from this work is still generally applicable, which is to tackle security from the ground up, starting of its definition, and to impose it through appropriate infrastructure.
Bibliography


[38] MULLINER, C. (2009) “Fuzzing the Phone in your Phone,” in Blackhat Conference, USA.

[39] 3RD GENERATION PARTNERSHIP PROJECT Technical realization of the Short Message Service (SMS), Tech. Rep. TS 03.40 v7.5.0, 3GPP.


Vita
Machigar Ongtang

Education

The Pennsylvania State University, University Park, PA, Ph.D. in Computer Science and Engineering, Summer 2010

University of Warwick, United Kingdom, M.Sc. in Information Technology for Manufacture with Distinction, 2004

Mahidol University, Thailand, B.Eng. in Computer Engineering with First Class Honors, 2002

Professional Experience

Research Assistant Penn State University, University Park, PA, Spring 2008 - Summer 2010. Researched into mobile phones and telecommunication security.

Research Assistant Penn State University, University Park, PA, Summer - Fall 2007. Researched into mobile agents and mobile multidatabases.

Teaching Assistant Penn State University, University Park, PA, Spring 2007. Assisted undergraduate class CSE497B - Computer Security.

Instructor Dhurakij Pundit University, Bangkok, Thailand, 10/2004 - 10/2005. Instructed undergraduate courses on data structure and algorithm, Java programming, web programming, and introduction to Information Technology (English program)

IT Staff University of Warwick, Coventry, United Kingdom, 03/2004 - 06/2004. Designed and developed Server Monitoring and Performance Monitoring System.

System Engineer NS Electronics, Bangkok, Thailand, 04/2002 - 05/2003. Administered Oracle Database (8.0.5/ 8.1.7/ 9i) system, administered production servers, and worked on system optimization projects.

Awards and Honors

Best Paper Award Annual Computer Security Applications Conference (AC-SAC), 2009.


Award for Performance in Science Studies The Professor Dr.Tab Nilanadhi Foundation, Chulalongkorn University, Thailand, 1999.