On Lightweight Mobile Phone Application Certification* †

William Enck, Machigar Ongtang, and Patrick McDaniel
Systems and Internet Infrastructure Security Laboratory
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
{enck,ongtang,mcdaniel}@cse.psu.edu

ABSTRACT

Users have begun downloading an increasingly large number of mobile phone applications in response to advancements in handsets and wireless networks. The increased number of applications results in a greater chance of installing Trojans and similar malware. In this paper, we propose the Kirin security service for Android, which performs lightweight certification of applications to mitigate malware at install time. Kirin certification uses security rules, which are templates designed to conservatively match undesirable properties in security configuration bundled with applications. We use a variant of security requirements engineering techniques to perform an in-depth security analysis of Android to produce a set of rules that match malware characteristics. In a sample of 311 of the most popular applications downloaded from the official Android Market, Kirin and our rules found 5 applications that implement dangerous functionality and therefore should be installed with extreme caution. Upon close inspection, another five applications asserted dangerous rights, but were within the scope of reasonable functional needs. These results indicate that security configuration bundled with Android applications provides practical means of detecting malware.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security

Keywords

mobile phone security, malware, Android

*The Kirin security service described in this paper is a malware focused revision of a similar system described in our previous work [10].

†This material is based upon work supported by the National Science Foundation under Grant No. CNS-0721579 and CNS-0643907. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'09, November 9–13, 2009, Chicago, Illinois, USA.
Copyright 2009 ACM 978-1-60558-352-5/09/11 ...$10.00.

1. INTRODUCTION

Mobile phones have emerged as a topic *du jour* for security research; however, the domain itself is still settling. Telecommunications technology is constantly evolving. It recently reached a critical mass with the widespread adoption of third generation (3G) wireless communication and handsets with advanced microprocessors. These capabilities provide the foundation for a new (and much anticipated) computing environment teeming with opportunity. Entrepreneurs have heavily invested in the mobile phone application market, with small fortunes seemingly made overnight. However, this windfall is not without consequence. The current mixture of information and accessibility provided by mobile phone applications seeds emerging business and social lifestyles, but it also opens opportunity to profit from users' misfortune.

To date, mobile phone malware has been primarily destructive and "proof-of-concept." However, Trojans such as Viver [19], which send SMS messages to premium rate numbers, indicate a change in malware motivations. Many expect mobile phone malware to begin following PC-based malware trends of fulfilling financial motivations [28]. Users are becoming more comfortable downloading and running mobile phone software. As this inevitably increases, so does the potential for user-installed malware.

The most effective phone malware mitigation strategy to date has been to ensure only approved software can be installed. Here, a certification authority (e.g., SymbianSigned, or Apple) devotes massive resources towards source code inspection. This technique can prevent both malware and general software misuse. For instance, software desired by the end user may be restricted by the service provider (e.g., VoIP and "Bluetooth tethering" applications). However, manual certification is imperfect. Malware authors have already succeeded in socially engineering approval [22]. In such cases, authorities must resort to standard revocation techniques.

We seek to mitigate malware and other software misuse on mobile phones without burdensome certification processes for each application. Instead, we perform lightweight certification at time of install using a set of predefined security rules. These rules decide whether or not the security configuration bundled with an application is safe. We focus our efforts on the Google-led Android platform, because it: 1) bundles useful security information with applications, 2) has been adopted by major US and European service providers, and 3) is open source.

In this paper, we propose the Kirin[1] security service for Android. Kirin provides practical lightweight certification of applications at install time. Achieving a practical solution requires overcoming multiple challenges. First, certifying applications based on security configuration requires a clear specification of undesirable proper-

[1]Kirin is the Japanese animal-god that protects the just and punishes the wicked.

ties. We turn to the field of security requirements engineering to design a process for identifying Kirin security rules. However, limitations of existing security enforcement in Android makes practical rules difficult to define. Second, we define a security language to encode these rules and formally define its semantics. Third, we design and implement the Kirin security service within the Android framework.

Kirin's practicality hinges on its ability to express security rules that simultaneously prevent malware and allow legitimate software. Adapting techniques from the requirements engineering, we construct detailed security rules to mitigate malware from an analysis of applications, phone stakeholders, and systems interfaces. We evaluate these rules against a subset of popular applications in the Android Market. 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). Note that this analysis flagged about 1.6% applications at install time as potentially dangerous. Thus, we show that even with conservative security policy, less than 1 in 50 applications needed any kind of involvement by phone users.

Kirin provides a practical approach towards mitigating malware and general software misuse in Android. In the design and evaluation of Kirin, this paper 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 rules to mitigate malware.* These rules are constructed purely from security configuration available in application package manifests.

The remainder of this paper proceeds as follows. Section 2 overviews the Kirin security service and software installer. Section 3 provides background information on mobile phone malware and the Android OS. Section 4 presents our rule identification process and sample security rules. Section 5 describes the Kirin Security Language and formally defines its semantics. Section 6 describes Kirin's implementation. Section 7 evaluates Kirin's practicality. Section 8 presents discovered vulnerabilities. Section 9 discusses related work. Section 10 concludes.

2. KIRIN OVERVIEW

The overwhelming number of existing malware requires manual installation by the user. While Bluetooth has provided the most effective distribution mechanism [28], as bulk data plans become more popular, so will SMS and email-based social engineering. Recently, Yxe [20] propagated via URLs sent in SMS messages. While application stores help 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.

Android's existing security framework restricts permission assignment to an application in two ways: user confirmation and signatures by developer keys. These permissions are referred to as "dangerous" and "signature" permissions, respectively (as discussed in Section 3.2). Android uses "signature" permissions to prevent third-party applications from inflicting harm to the phone's trusted computing base.

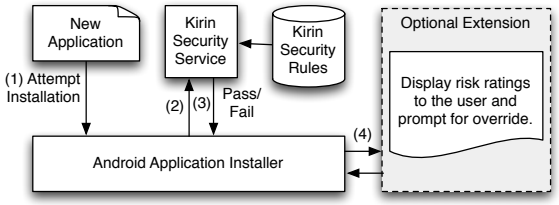

Figure 1: Kirin based software installer

The Open Handset Alliance (Android's founding entity) proclaims the mantra, "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 trusted computing base should remain fixed and must be protected. "Signature" permissions protect particularly dangerous functionality. However, there is a tradeoff 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." However, 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 (discussed further in Section 4).

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 security rules. The rules represent templates of undesirable security properties. Alone, these properties do not necessarily indicate malicious potential; however, as we describe in Section 4, specific combinations allow malfeasance. For example, an application that can start on boot, read geographic location, and access the Internet is potentially a tracker installed as premeditated spyware (a class of malware discussed in Section 3.1). 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 1 depicts the Kirin based software installer. The installer first extracts security configuration from the target package manifest. Next, the Kirin security service evaluates the configuration against a collection of security rules. 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 PrivacyBird [7] (PrivacyBird is a web browser plug-in 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. This paper focuses specifically on *identifying* potential harmful configurations and leaves the rating system for future work.

3. BACKGROUND INFORMATION

Kirin relies on well constructed security rules to be effective. Defining security rules for Kirin requires a thorough understanding of threats and existing protection mechanisms. This section begins by discussing past mobile phone malware and projects classifications for future phone malware based on trends seen on PCs. We then provide an overview of Android's application and security frameworks.

3.1 Mobile Phone Threats

The first mobile phone virus was observed in 2004. While Cabir [12] carries a benign payload, it demonstrated the effectiveness of Bluetooth as a propagation vector. The most notable outbreak was at the 2005 World Championships in Athletics [21]. More interestingly, Cabir did not exploit any vulnerabilities. It operated entirely within the security parameters of both its infected host (Symbian OS) and Bluetooth. Instead, it leveraged flaws in the user interface. While a victim is in range, Cabir continually sends file transfer requests. When the user chooses "no," another request promptly appears, frustrating the user who subsequently answers "yes" repeatedly in an effort to use the phone [28].

Cabir was followed by a series of viruses and Trojans targeting the Symbian Series 60 platform, each increasing in complexity and features. Based on Cabir, Lasco [16] additionally infects all available software package (SIS) files residing on the phone on the assumption that the user might share them. Commwarrior [14] added MMS propagation in addition to Bluetooth. Early variants of Commwarrior attempt to replicate via Bluetooth between 8am and midnight (when the user is mobile) and via MMS between midnight and 7am (when the user will not see error messages resulting from sending an MMS to non-mobile devices). Originally masquerading as a theme manager, the Skulls [18] Trojan provided one of the first destructive payloads. When installed, Skulls writes non-functioning versions of all applications to the `c:` drive, overriding identically named files in the firmware ROM `z:` drive. All applications are rendered useless and their icons are replaced with a skull and crossbones. Other Trojans, e.g., Drever [15], fight back by disabling Antivirus software. The Cardblock [13] Trojan embeds itself within a pirated copy of InstantSis (a utility to extract SIS software packages from a phone). However, Cardblock sets a random password on the phone's removable memory card, making the user's data inaccessible.

To date, most phone malware has been either "proof-of-concept" or destructive, a characteristic often noted as resembling early PC malware. Recent PC malware more commonly scavenges for valuable information (e.g., passwords, address books) or joins a botnet [42]. The latter frequently enables denial of service (DoS)-based extortion. It is strongly believed that mobile phone malware will move in similar directions [8, 28]. In fact, Pbstealer [17] already sends a user's address book to nearby Bluetooth devices, and Viver [19] sends SMS messages to premium-rate numbers, providing the malware writer with direct monetary income.

Mobile phone literature has categorized phone malware from different perspectives. Guo et al. [25] consider categories of resulting network attacks. Cheng et al. [6] derive models based on infection vector (e.g., Bluetooth vs. MMS). However, we find a taxonomy based on an attacker's motivations [8] to be the most useful when designing security rules for Kirin. We foresee the following motivations seeding future malware (the list is not intended to be exhaustive):

- *Proof-of-concept*: Such malware often emerges as new infection vectors are explored by malware writers and frequently have unintended consequences. For example, Cabir demonstrated Bluetooth-based distribution and inadvertently drained device batteries. Additionally, as non-Symbian phones gain stronger user bases (Symbian market share dropped 21% between August 2008 and February 2009 [1] in response to the iPhone), proof-of-concept malware will emerge for these platforms.
- *Destructive*: Malware such as Skulls and Cardblock (described above) 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. 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*: FlexiSPY (`www.flexispy.com`) is marketed as a tool to "catch cheating spouses" and is available for Symbian, Windows Mobile, and BlackBerry. 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*: Viver (described above) directly compensates the malware's author by sending messages to premium SMS numbers. We will undoubtedly see similar malware appearing more frequently. 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 [3].
- *Information scavengers*: Web-based malware currently scours PCs for valuable address books and login credentials (e.g., usernames, passwords, and cookies for two-factor authentication for bank websites) [42]. Mobile phones are much more organized then their PC counterparts, making them better targets for such malware [8]. 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 [8].
- *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) [23]. Traynor predicts the existence of a mobile botnet in 2009 [24]. 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

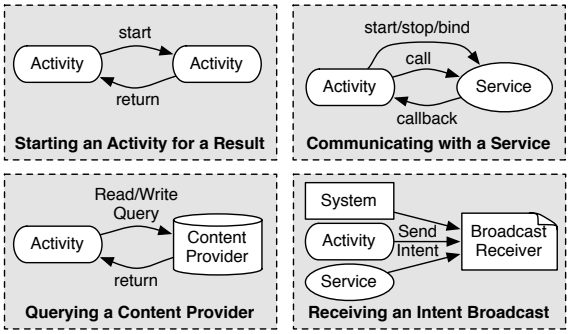

Figure 2: Typical IPC between application components

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 wide spread. Finally, the phone functionality will be used. For example, telemarketers could use automated dialers from mobots to distribute advertisements, creating "voice-spam" [46].

3.2 The Android Operating System

Android is an operating system for mobile phones. However, it is better described as a middleware running on top of embedded Linux. The underlying Linux internals have been customized to provide strong isolation and contain exploitation. Each application is written in Java and runs as a process with a unique UNIX user identity. This design choice minimizes the effects of a buffer overflows. For example, a vulnerability in web browser libraries [29] allowed an exploit to take control of the web browser, but the system and all other applications remained unaffected.

All inter-application communication passes through Android's middleware (with a few exceptions). In the remainder of this section, we describe how the middleware operates and enforces security. For brevity, we concentrate on the concepts and details necessary to understand Kirin. Enck et al. [11] provide a more detailed description of Android's security model with helpful examples.

3.2.1 Application Structure

The Android middleware defines four types of inter-process communication (IPC).[2] The types of IPC directly correspond to the four types of *components* that make up applications. Generally, IPC takes the form of an "Intent message". Intents are either addressed directly to a component using the application's unique namespace, or more commonly, to an "action string." Developers specify "Intent filters" based on action strings for components to automatically start on corresponding events. Figure 2 depicts typical IPC between components that potentially crosses applications.

- An *Activity* component interfaces with the physical user via the touchscreen and keypad. Applications commonly contain many Activities, one for each "screen" presented to the user. The interface progression is a sequence of one Activity "starting" another, possibly expecting a return value. Only one Activity on the phone has input and processing focus at a time.
- A *Service* component provides background processing that continues even after its application loses focus. Services also define arbitrary interfaces for remote procedure call (RPC), including method execution and callbacks, which can only be called after the service has been "bound".
- A *Content Provider* component is a database-like mechanism for sharing data with other applications. The interface does not use Intents, but rather is addressed via a "content URI." It supports standard SQL-like queries, e.g., `SELECT`, `UPDATE`, `INSERT`, through which components in other applications can retrieve and store data according to the Content Provider's schema (e.g., an address book).
- A *Broadcast Receiver* component is an asynchronous event mailbox for Intent messages "broadcasted" to an action string. Android defines many standard action strings corresponding to system events (e.g., the system has booted). Developers often define their own action strings.

Every application package includes a manifest file. The manifest specifies all components in an application, including their types and Intent filters. Note that Android 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 register them. The manifest also includes security information, discussed next.

3.2.2 Security Enforcement

Android's middleware mediates IPC based on *permission labels* using a user space reference monitor [2]. For the most part, security decisions are statically defined by the applications' package manifests. Security policy in the package manifest primarily consists of 1) permission labels used (requested) by the application, and 2) a permission label to restrict access to each component. When an application is installed, Android decides whether or not to grant (assign) the permissions requested by the application. Once installed, this security policy cannot change.

Permission labels map the ability for an application to perform IPC to the restriction of IPC at the target interface. Security policy decisions occur on the granularity of applications. Put simply, *an application may initiate IPC with a component in another (or the same) application if it has been assigned the permission label specified to restrict access to the target component IPC interface.* Permission labels are also used to restrict access to certain library APIs. For instance, there is a permission label that is required for an application to access the Internet. Android defines many permission labels to protect libraries and components in core applications. However, applications can define their own.

There are many subtleties when working with Android security policy. First, not all policy is specified in the manifest file. The API for broadcasting Intents optionally allows the developer to specify a permission label to restrict which applications may receive it. This provides an access control check in the reverse direction. Additionally, the Android API includes a method to arbitrarily insert a reference monitor hook anywhere in an application. This feature is primarily used to provide differentiated access to RPC interfaces in a Service component. Second, the developer is not forced to specify a permission label to restrict access to a component. If no label is specified, there is no restriction (i.e., default allow). Third, components can be made "private," precluding them from access by other applications. The developer need not worry about specifying permission labels to restrict access to private components. Fourth, developers can specify separate permission labels to restrict access to the read and write interfaces of a Content Provider component. Fifth, developers can create "Pending Intent" objects that can be passed to other applications. That application can fill in both data and address fields in the Intent message. When the Pending In-

[2]Unless otherwise specified, we use "IPC" to refer to the IPC types specifically defined by the Android middleware, which is distinct from the underlying Linux IPC.

tent is sent to the target component, it does so with the permissions granted to the original application. Sixth, in certain situations, an application can delegate its access to subparts (e.g., records) of a Content Provider. These last two subtleties add discretion to an otherwise mandatory access control (MAC) system.

Most of Android's core functionality is implemented as separate applications. For instance, the "Phone" application provides voice call functionality, and the "MMS" application provides a user interface for sending and receiving SMS and MMS messages. Android protects these applications in the same way third-party developers protect their applications. We include these core applications when discussing Android's trusted computing base (TCB). A final subtlety of Android's security framework relates to how applications are granted the permission labels they request. There are three main "protection levels" for permission labels: a "normal" permission is granted to any application that requests it; a "dangerous" permission is only granted after user approval at install-time; and a "signature" permission is only granted to applications signed by the same developer key as the application defining the permission label. This last protection level is integral in ensuring third-party applications do not gain access affecting the TCB's integrity.

4. KIRIN SECURITY RULES

The malware threats and the Android architecture introduced in the previous sections serve as the background for developing Kirin security rules to detect potentially dangerous application configurations. To ensure the security of a phone, we need a clear definition of a secure phone. Specifically, we seek to define the conditions that an application must satisfy for a phone to be considered safe. To define this concept for Android, we turn to the field of security requirements engineering, which is an off-shoot of requirements engineering and security engineering. The former is a well-known fundamental component of software engineering in which business goals are integrated with the design. The latter focuses on the threats facing a specific system.

Security requirements engineering is based upon three basic concepts. 1) *functional requirements* define how a system is supposed to operate in normal environment. For instance, when a web browser requests a page from a web server, the web server returns the data corresponding to that file. 2) *assets* are "...entities that someone places value upon" [31]. The webpage is an asset in the previous example. 3) *security requirements* are "...constraints on functional requirements to protect the assets from threats" [26]. For example, the webpage sent by the web server must be identical to the webpage received by the client (i.e., integrity).

The security requirements engineering process is generally systematic; however, it requires a certain level of human interaction. Many techniques have been proposed, including SQUARE [5, 34], SREP [35, 36], CLASP [40], misuse cases [33, 47], and security patterns [27, 45, 48]. Related implementations have seen great success in practice, e.g., Microsoft uses the Security Development Lifecycle (SDL) for the development of their software that must withstand attacks [32], and Oracle has developed OSSA for the secure software development of their products [41].

Commonly, security requirements engineering begins by creating functional requirements. This usually involves interviewing stakeholders [5]. Next, the functional requirements are translated into a visual representation to describe relationships between elements. Popular representations include use cases [47] and context diagrams using problem frames [37, 26]. Based on these requirements, assets are identified. Finally, each asset is considered with respect to high level security goals (e.g., confidentiality, integrity, and availability). The results are the security requirements.

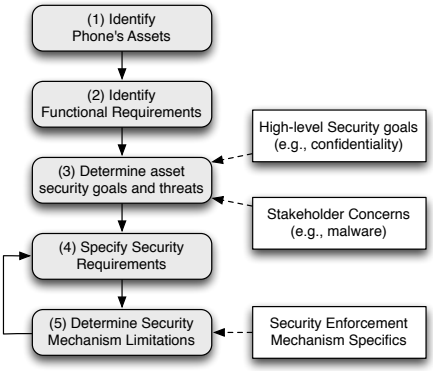

Figure 3: Procedure for requirements identification

Unfortunately, we cannot directly utilize these existing techniques because they are designed to supplement system and software development. Conversely, we wish to retrofit security requirements on an existing design. There is no clearly defined usage model or functional requirements specification associated with the Android platform or the applications. Hence, we provide an adapted procedure for identifying security requirements for Android. *The resulting requirements directly serve as Kirin security rules.*

4.1 Identifying Security Requirements

We use existing security requirements engineering techniques as a reference for identifying dangerous application configurations in Android. Figure 3 depicts our procedure, which consists of five main activities.

Step 1: Identify Assets.

Instead of identifying assets from functional requirements, we extract them from the features on the Android platform. Google has identified many assets already in the form of permission labels protecting resources. Moreover, as the broadcasted Intent messages (e.g. those sent by the system) impact both platform and application operation, they are assets. Lastly, all components (Activities, etc.) of system applications are assets. While they are not necessarily protected by permission labels, many applications call upon them to operate.

As an example, Android defines the `RECORD_AUDIO` permission to protect its audio recorder. Here, we consider the asset to be microphone input, as it records the user's voice during phone conversations. Android also defines permissions for making phone calls and observing when the phone state changes. Hence, call activity is an asset.

Step 2: Identify Functional Requirements.

Next, we carefully study each asset to specify corresponding functional descriptions. These descriptions indicate how the asset interacts with the rest of the phone and third-party applications. This step is vital to our design, because both assets and functional descriptions are necessary to investigate realistic threats.

Continuing the assets identified above, when the user receives an incoming call, the system broadcasts an Intent to the `PHONE_STATE` action string. It also notifies any applications that have registered a `PhoneStateListener` with the system. The same notifications are sent on outgoing call. Another Intent to the `NEW_OUTGOING_CALL` action string is also broadcasted. Furthermore, this additional broadcast uses the "ordered" option, which serializes the broadcast and allows any recipient to cancel it. If this occurs, subsequent Broadcast Receivers will not receive the Intent mes-

sage. This feature allows, for example, an application to redirect international calls to the number for a calling card. Finally, audio can be recorded using the `MediaRecorder` API.

Step 3: Determine Assets Security Goals and Threats.

In general, security requirements engineering considers high level security goals such as confidentiality, integrity, and availability. For each asset, we must determine which (if not all) goals are appropriate. Next, we consider how the functional requirements can be abused with respect to the remaining security goals. Abuse cases that violate the security goals provide *threat descriptions*. We use the malware motivations described in Section 3.1 to motivate our threats. Note that defining threat descriptions sometimes requires a level of creativity. However, trained security experts will find most threats straightforward after defining the functional requirements.

Continuing our example, we focus on the confidentiality of the microphone input and phone state notifications. These goals are abused if a malware records audio during voice call and transmits it over the Internet (i.e., premeditated spyware). 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.*"

Step 4: Develop Asset's Security Requirements.

Next, we define security requirements from the threat descriptions. Recall from our earlier discussion, security requirements are constraints on functional requirements. That is, they specify who can exercise functionality or conditions under which functionality may occur. Frequently, this process consists of determining which sets of functionality are required to compromise a threat. The requirement is the security rule that restricts the ability for this functionality to be exercised in concert.

We observe that the eavesdropper requires a) notification of an incoming or outgoing call, b) the ability to record audio, and c) access to the Internet. Therefore, our security requirement, which acts as Kirin security rule, becomes, "*an application must not be able to receive phone state, record audio, and access the Internet.*"

Step 5: Determine Security Mechanism Limitations.

Our final step caters to the practical limitations of our intended enforcement mechanism. Our goal is to identify potentially dangerous configurations at install time. Therefore, we cannot ensure runtime support beyond what Android already provides. Additionally, we are limited to the information available in an application package manifest. For both these reasons, we must refine our list of security requirements (i.e., Kirin 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 [30], because Android does not support history-based policies. Security rules must also be translated to be expressed in terms of 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, the iteration between Steps 4 and 5 is required to adjust the rules to work within our limitations. Additionally, security rules can be subdivided to be more straightforward.

The permission labels corresponding to the restricted functionality in our running example include `READ_PHONE_STATE`, `PROCESS_OUTGOING_CALLS`, `RECORD_AUDIO`, and `INTERNET`. Furthermore, we subdivide our security rule to remove the disjunctive logic resulting from multiple ways for the eavesdropper to be notified of voice call activity. Hence, we create the following adjusted security rules: a) "*an application must not have the `READ_PHONE_STATE`, `RECORD_AUDIO`, and `INTERNET` permissions.*" and the nearly identical b) "*an application must not have the `PROCESS_OUTGOING_CALLS`, `RECORD_AUDIO`, and `INTERNET` permissions.*"

4.2 Sample Malware Mitigation Rules

The remainder of this section discusses Kirin security rules we developed following our 5-step methodology. For readability and ease of exposition, we have enumerated the precise security rules in Figure 4. We refer to the rules by the indicated numbers for the remainder of the paper. We loosely categorize Kirin security rules by their complexity.

4.2.1 Single Permission Security Rules

Recall that a number of Android's "dangerous" permissions may be "too dangerous" for some production environments. 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 available documentation). The corresponding API is "hidden" in the most recent SDK environment (at the time of writing, version 1.1r1). The hidden APIs are not accessible by third-party applications but only by system applications. However, hidden APIs are no substitute for security. 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 made for other permission labels protecting hidden APIs (e.g., Bluetooth APIs not yet considered mature enough for general use).

4.2.2 Multiple Permission Security Rules

Voice and location eavesdropping malware requires permissions to record audio and access location information. However, legitimate applications use these permissions as well. Therefore, we must define rules with respect to multiple permissions. To do this, we consider the minimal set of functionality required to compromise a threat. Rules 2 and 3 protect against the voice call eavesdropper used as a running example in Section 4.1. Similarly, Rules 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" permission (and hence is always granted). However, the permission label provides valuable insight into the functional requirements of an application. In general, *Kirin security rules are more expressible as the number of available permission labels increases.*

Rules 6 and 7 consider malware's interaction with SMS. Rule 6 protects against malware 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

(1) *An application must not have the* SET_DEBUG_APP *permission label.*

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

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

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

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

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

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

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

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

Figure 4: Sample Kirin security rules to mitigate malware

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. Similar to Rule 6, this rule ensures 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 uninstalling 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.

4.2.3 Permission and Interface Security Rules

Permissions alone are not always enough to characterize malware behavior. Rule 9 provides 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. However, if an application has the SET_PREFERRED_APPLICATION permission label, 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 trusted computing base.

5. KIRIN SECURITY LANGUAGE

We now describe the Kirin Security Language (KSL) to encode security rules for the Kirin security service. Kirin uses an application's package manifest as input. The rules identified in Section 4 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.

5.1 KSL Syntax

Figure 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.

$$\langle\text{rule-set}\rangle ::= \langle\text{rule}\rangle \mid \langle\text{rule}\rangle\ \langle\text{rule-set}\rangle \quad (1)$$

$$\langle\text{rule}\rangle ::= \text{“}\texttt{restrict}\text{”}\ \langle\text{restrict-list}\rangle \quad (2)$$

$$\langle\text{restrict-list}\rangle ::= \langle\text{restrict}\rangle \mid \langle\text{restrict}\rangle\ \text{“}\texttt{and}\text{”}\ \langle\text{restrict-list}\rangle \quad (3)$$

$$\langle\text{restrict}\rangle ::= \text{“}\texttt{permission [}\text{”}\ \langle\text{const-list}\rangle\ \text{“}\texttt{]}\text{”} \mid \text{“}\texttt{receive [}\text{”}\ \langle\text{const-list}\rangle\ \text{“}\texttt{]}\text{”} \quad (4)$$

$$\langle\text{const-list}\rangle ::= \langle\text{const}\rangle \mid \langle\text{const}\rangle\ \text{“}\texttt{,}\text{”}\ \langle\text{const-list}\rangle \quad (5)$$

$$\langle\text{const}\rangle ::= \text{“}\texttt{'}\text{”}\texttt{[A-Za-z0-9_.]+}\text{“}\texttt{'}\text{”} \quad (6)$$

Figure 5: KSL syntax in BNF.

5.2 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 $\mathcal{P}$ be the set of possible permission labels and $\mathcal{A}$ be the set of possible action strings used by Activities, Services, and Broadcast Receivers to receive Intents. Then, each rule $r_i \in \mathcal{R}$ is a tuple $(2^{\mathcal{P}}, 2^{\mathcal{A}})$.[3] 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 $P_i \in 2^{\mathcal{P}}$ and $A_i \in 2^{\mathcal{A}}$.

Let $R \subseteq \mathcal{R}$ 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 $\mathcal{C}$ 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 7). We define configuration $c \in \mathcal{C}$ as a tuple $(2^{\mathcal{P}}, 2^{\mathcal{A}})$. 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 $P_t \in 2^{\mathcal{P}}$ and $A_t \in 2^{\mathcal{A}}$.

[3]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 now define the semantics of a set of KSL rules. Let $fail : \mathcal{C} \times \mathcal{R} \rightarrow \{\texttt{true}, \texttt{false}\}$ 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:

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

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

Let $F_R : \mathcal{C} \rightarrow \mathcal{R}$ be a function returning the set of all rules in $R \in 2^{\mathcal{R}}$ 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 2.

6. KIRIN SECURITY SERVICE

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 software installer. This service reads KSL rules 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 KSL rules. 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.

7. EVALUATION

Practical security rules must both mitigate malware and allow legitimate applications to be installed. Section 4 argued that our sample security rules can detect specific types of malware. However, Kirin's certification technique conservatively detects dangerous functionality, and 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 reduce this number further.

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 5.

7.1 Empirical Results

Our analysis tested all 311 applications against the security rules listed in Figure 4. 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 1: Applications failing Rule 2

Application	Description
Walki Talkie Push to Talk	Walkie-Talkie style voice communication.
Shazam	Utility to identify music tracks.
Inauguration Report	Collaborative journalism application.

Table 1 lists the applications that fail Rule 2. Recall that Rule 2 defends against 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 1 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 2 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 2, 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.

The remaining four applications in Table 2 result from the limitations in Kirin's input. That is, Kirin cannot inspect how an ap-

Table 2: Applications failing Rule 4 and 5

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

* Did not fail Rule 5

plication 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.

7.2 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 verses 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.

Finally, Kirin's expressibility is restricted by the policy that Android enforces. Android policy itself is static and does not support runtime logic. Therefore, it cannot enforce that no more than 10 SMS messages are sent per hour [30]. However, this is a limitation of Android and not Kirin.

8. DISCOVERED VULNERABILITIES

The process of retrofitting security requirements for Android had secondary effects. In addition to identifying rules for Kirin, we discovered a number of configuration and implementation flaws. Step 1 identifies assets. However, 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. As we continued our investigation with the most recent version of Android (v1.1r1), we discovered a number of APIs do not check permissions as prescribed in the documentation. 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.

9. RELATED WORK

The best defense for mobile phone malware is still unclear. Most likely, it requires a combination of solutions. Operating systems protections have been improving. The usability flaw allowing Cabir to effectively propagate has been fixed, and Symbian 3rd Edition uses Symbian Signed (`www.symbiansigned.com`), which ensures some amount of software vetting before an application can be installed. While, arguably, Symbian Signed only provides weak

security (socially engineered signatures have occurred [22]), it provides more protection than previous platform versions. Unfortunately, some users disable it.

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 [4] 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 [6, 44]. 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.

Preventative techniques have also been proposed. Muthukumaran et al. [38] extend Openmoko with SELinux policies to isolate untrusted software. Zhang et al. [49] incorporate trusted computing and SELinux into mobile phones. Security-by-contract [9] retrofits Microsoft's compact .NET platform by associating an application with a "contract" of declared functionality. If the application deviates from the contractual policy, the runtime environment interrupts execution. Kirin supplements Android's existing security infrastructure. It infers application functionality from security configuration available in the package manifest. While Android's runtime security enforcement is less expressive than security-by-contract, it is significantly lighter-weight, and Kirin provides no additional runtime overhead.

Outside the domain of mobile phones, others have looked at certifying applications containing security configuration. Rueda et al. [43] extract security policy from Jif applications and test compliance with SELinux policy. Similarly, proof carry code [39] provides a mechanism for a platform to check if an application behaves in an expected way.

10. CONCLUSION

As users become more comfortable downloading software for mobile phones, malware targeting phones will increase. Kirin provides lightweight certification at install time that does not require burdensome code inspection for each application. We have shown that Kirin can express meaningful security rules to mitigate 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 to the security requirements engineering process to discover additional rules to defend against malware.

This work is simply the first step in a longer journey towards realizing practical mobile phone security. We plan to extend our analysis of security configuration included in package manifests by enhancing the development environment to ensure applications are distributed with proper protections.

Acknowledgements

We would like to thank Kevin Butler, Stephen McLaughlin, and Trent Jaeger for their invaluable comments during the design and formulation of this work. We also thank Adam Smith for his feedback while designing the logic. Finally, we thank the SIIS lab as a whole for their continual feedback during the writing of this paper.

11. REFERENCES

[1] AdMob. AdMob Mobile Metrics Report. http://www.admob.com/marketing/pdf/mobile_metrics_feb_09.pdf, February 2009.

[2] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, The Mitre Corporation, Air Force Systems Division, Hanscom AFB, Badford, MA, October 1972.

[3] Apple, Inc. Get Ready for iPhone OS 3.0. http://developer.apple.com/iphone/program/sdk.html. Accessed April 2009.

[4] A. Bose, X. Hu, K. G. Shin, and T. Park. Behavioral Detection of Malware on Mobile Handsets. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 225–238, 2008.

[5] P. Chen, M. Dean, D. Ojoko-Adams, H. Osman, L. Lopez, and N. Xie. System Quality Requirements Engineering (SQUARE) Methodology: Case Study on Asset Management System. Technical Report CMU/SEI-2004-SR-015, Software Engineering Institute, Carnegie Mellon University, December 2004.

[6] J. Cheng, S. H. Wong, H. Yang, and S. Lu. SmartSiren: Virus Detection and Alert for Smartphones. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 258–271, June 2007.

[7] L. F. Cranor. P3P: Making Privacy Policies More Useful. *IEEE Security & Privacy magazine*, 1(6):50–55, November-December 2003.

[8] D. Dagon, T. Martin, and T. Starner. Mobile Phones as Computing Devices: The Viruses are Coming! *IEEE Pervasive Computing*, 3(4):11–15, October-December 2004.

[9] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-contract on the .NET platform. *Information Security Technical Report*, 13(1):25–32, January 2008.

[10] W. Enck, M. Ongtang, and P. McDaniel. Mitigating Android Software Misuse Before It Happens. Technical Report NAS-TR-0094-2008, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, November 2008.

[11] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android Security. *IEEE Security & Privacy Magazine*, 7(1):50–57, January/February 2009.

[12] F-Secure Corporation. Virus Description: Cabir. http://www.f-secure.com/v-descs/cabir.shtml. Accessed March 2009.

[13] F-Secure Corporation. Virus Description: Cardblock.A. http://www.f-secure.com/v-descs/cardblock_a.shtml. Accessed March 2009.

[14] F-Secure Corporation. Virus Description: Commwarrior. http://www.f-secure.com/v-descs/commwarrior.shtml. Accessed March 2009.

[15] F-Secure Corporation. Virus Description: Drever.A. http://www.f-secure.com/v-descs/drever_a.shtml. Accessed March 2009.

[16] F-Secure Corporation. Virus Description: Lasco.A. http://www.f-secure.com/v-descs/lasco_a.shtml. Accessed March 2009.

[17] F-Secure Corporation. Virus Description: Pbstealer.A. http://www.f-secure.com/v-descs/pbstealer_a.shtml. Accessed March 2009.

[18] F-Secure Corporation. Virus Description: Skulls.A. http://www.f-secure.com/v-descs/skulls.shtml. Accessed March 2009.

[19] F-Secure Corporation. Virus Description: Viver.A. http://www.f-secure.com/v-descs/trojan_symbos_viver_a.shtml. Accessed March 2009.

[20] F-Secure Corporation. Virus Description: Worm:SymbOS/Yxe.A. http://www.f-secure.com/v-descs/worm_symbos_yxe.shtml. Accessed March 2009.

[21] F-Secure Corporation. TeliaSonera Finland and F-Secure united against mobile viruses: Cabir spreads at the World Championships. http://www.f-secure.com/en_EMEA/about-us/pressroom/news/2005/fs_news_20050811_1_eng.html, August 2005.

[22] F-Secure Corporation. Just because it's Signed doesn't mean it isn't spying on you. http://www.f-secure.com/weblog/archives/00001190.html, May 2007.

[23] C. Fleizach, M. Liljenstam, P. Johansson, G. M. Moelker, and A. Méhes. Can you Infect Me Now? Malware Propagation in Mobile Phone Networks. In *Proceedings of the ACM Workshop On Rapid Malcode (WORM)*, pages 61–68, Novermber 2007.

[24] Georgia Tech Information Security Center. Emerging Cyber Threats Report for 2009. http://www.gtisc.gatech.edu/pdf/CyberThreatsReport2009.pdf, October 2008.

[25] C. Guo, H. J. Wang, and W. Zhu. Smart-Phone Attacks and Defenses. In *Third Workshop on Hot Topics in Networks (HotNets)*, November 2004.

[26] C. B. Haley, J. D. Moffett, R. Laney, and B. Nuseibeh. A framework for security requirements engineering. In *SESS '06: Proceedings of the 2006 international workshop on Software engineering for secure systems*, pages 35–42, 2006.

[27] D. Hatebur, M. Heisel, and H. Schmidt. A Pattern System for Security Requirements Engineering. In *ARES '07: Proceedings of the The Second International Conference on Availability, Reliability and Security*, pages 356–365, 2007.

[28] M. Hyppönen. Mobile Malware. USENIX Security Symposium, August 2007. Invited Talk.

[29] Independent Security Evaluators. Exploiting Android. http://securityevaluators.com/content/case-studies/android/index.jsp.

[30] I. Ion, B. Dragovic, and B. Crispo. Extending the Java Virtual Machine to Enforce Fine-Grained Security Policies in Mobile Devices. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, December 2007.

[31] *ISO/IEC 15408-1 Information technology - Security techniques - Evaluation criteria for IT security - Part 1: Introduction and general model*, October 2005.

[32] S. Lipner and M. Howard. The Trustworthy Computing Security Development Lifecycle. http://msdn.microsoft.com/en-us/library/ms995349.aspx, March 2005.

[33] J. McDermott and C. Fox. Using Abuse Case Models for Security Requirements Analysis. In *Proceedings of the 15th Annual Computer Security Applications Conference*, 1999.

[34] N. R. Mead. How To Compare the Security Quality Requirements Engineering (SQUARE) Method with Other Methods. Technical Report CMU/SEI-2007-TN-021, Software Engineering Institute, Carnegie Mellon University, August 2007.

[35] D. Mellado, E. Fernández-Medina, and M. Piattini. Applying a Security Requirements Engineering Process. *LNCS: Computer Security – ESORICS 2006*, 4189/2006:192–206, September 2006.

[36] D. Mellado, E. Fernández-Medina, and M. Piattini. A common criteria based security requirements engineering process for the development of secure information systems. *Computer Standards & Interfaces*, 29(2):244–253, 2007.

[37] J. D. Moffett, C. B. Haley, and B. Nuseibeh. Core Security Requirements Artefacts. Technical report, Open University, UK, 2004.

[38] D. Muthukumaran, A. Sawani, J. Schiffman, B. M. Jung, and T. Jaeger. Measuring Integrity on Mobile Phone Systems. In *Proceedings of the ACM Symposium on Access Control Models and Technologies*, pages 155–164, June 2008.

[39] G. C. Necula. Proof-Carrying Code. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 106–119, Jan. 1997.

[40] Open Web Application Security Project (OWASP) Foundation. CLASP (Comprehensive, Lightweight Application Security Process) Project. http://www.owasp.org/index.php/OWASP_CLASP_Project, March 2009.

[41] Oracle. Oracle Software Security Assurance. http://www.oracle.com/security/software-security-assurance.html, April 2009.

[42] M. Polychronakis, P. Mavrommatis, and N. Provos. Ghost Turns Zombie: Exploring the Life Cycle of Web-based Malware. In *Proceedings of the USENIX Workshop on Large-Scale and Emergent Threats (LEET)*, April 2008.

[43] S. Rueda, D. King, and T. Jaeger. Verifying Compliance of Trusted Programs. In *Proceedings of the USENIX Security Symposium*, pages 321–334, August 2008.

[44] A.-D. Schmidt, F. Peters, F. Lamour, C. Scheel, S. A. Camtepe, and S. Albayrak. Monitoring Smartphones for Anomaly Detection. *Mobile Networks and Applications*, 14(1):92–106, February 2009.

[45] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. Wiley, 2009.

[46] D. Shin, J. Ahn, and C. Shim. Progressive Multi Gray-Leveling: A Voice Spam Protection Algorithm. *IEEE Network*, 20(5):18–24, September-October 2006.

[47] G. Sindre and A. L. Opdahl. Eliciting security requirements with misuse cases. *LNCS: Requirements Engineering*, pages 34–44, June 2004.

[48] N. Yoshioka, H. Washizaki, and K. Maruyama. A survey on security patterns. *Progress in Informatics, Special issue: The future of software engineering for security and privacy*, 5:35–47, October 2008.

[49] X. Zhang, O. Aciiçmez, and J.-P. Seifert. A Trusted Mobile Phone Reference Architecture via Secure Kernel. In *Proceedings of the ACM workshop on Scalable Trusted Computing*, pages 7–14, November 2007.