Automated Remote Repair for Mobile Malware

Yacin Nadji, Jonathon Giffin, and Patrick Traynor
School of Computer Science, Georgia Institute of Technology
{yacin.nadji, giffin, traynor}@cc.gatech.edu

ABSTRACT

Mobile application markets currently serve as the main line of defense against malicious applications. While marketplace revocations have successfully removed the few overtly malicious applications installed on mobile devices, the anticipated coming flood of mobile malware mandates the need for mechanisms that can respond faster than manual intervention. In this paper, we propose an infrastructure that automatically identifies and responds to malicious mobile applications based on their network behavior. We design and implement a prototype, Airmid, that uses cooperation between in-network sensors and smart devices to identify the provenance of malicious traffic. We then develop sample malicious mobile applications exceeding the capabilities of malware recently discovered in the wild, demonstrate the ease with which they can evade current detection techniques, and then use Airmid to show a range of automated recovery responses ranging from on-device firewalling to application removal.

1. INTRODUCTION

Malware infections on mobile phones have flourished in recent years. Nearly immediately after the introduction of functionality beyond basic telephony, researchers have suggested that such devices were likely targets of malicious software [13, 23]. However, even as mobile operating systems move towards greater sophistication and an increasing number of malicious applications have been discovered, large-scale infection has been avoided. This relative safety can be largely attributed to the revocation capabilities of mobile application markets which, upon discovering or being notified of the existence of a malicious program, can remove it from both the marketplace and installed platforms.

Manually-triggered revocations have been successful because the number of malicious applications has been small. As the rate with which new mobile malware is discovered begins to approach that of traditional malware, estimated by some as greater than 70,000 new samples per day [37], such mechanisms are unlikely to be able to continue to respond rapidly. Specifically, cellular providers will not be able to rely *solely* upon the rapid identification and removal of malware by mobile market operators.

In this paper, we propose, design, and implement Airmid[1], a system for the automated detection of and response to malicious software infections on handheld mobile devices. Our architecture leverages cooperation between network monitors and on-device software components. We place the main detection component within the network so that no malware-specific knowledge needs to be stored at every mobile device. When the network sensor detects malicious traffic (which can be done using traditional security tools including DNS blacklists, domain firewall policies, an IDS, etc), it notifies devices sending or receiving that traffic via an authenticated channel. A protected software agent on each device then identifies executing processes responsible for the malicious traffic and initiates a recovery action to repair the infection: traffic can be filtered at the device, apps can be sandboxed or deleted, or the device itself can be patched or restored to its factory defaults. This system allows malware detection and response to occur at machine speed without human interaction and *without burdening small handheld systems with the computation, storage, and power consumption typical of traditional anti-virus systems.*

As in desktop malware, we expect malicious apps to increase in sophistication as defensive utilities become widespread; however, current malicious applications remain rather simplistic. To facilitate testing with complex malicious apps, we developed laboratory samples of mobile malware exhibiting characteristics common to mature desktop-class malware. The basic functionality of the malware reflects attacks already occuring in cellular devices: our apps leak private data [38], dial premium numbers [25], and participate in botnet activity [35]. To this we added complex evasive functionality: our samples detect the presence of an emulated environment and change their behavior, create hidden background processes, scrub logs, and restart on reboot. These samples demonstrate that mobile device software architectures permit the creation of advanced malware that cannot be easily identified prior to distribution, underscoring the need for rapid recovery after infection. For safety, our malware lacks propagation ability and was tested only on a closed network.

In summary, we make the following contributions:

- **Identification of current remediation shortcom-**

[1]Airmid is the Celtic goddess of healing, and is known for her ability to bring the dead back to life.

ACSAC '11 Dec. 5-9, 2011, Orlando, Florida USA

ings: While mobile application markets have thus far successfully removed most discovered malware, we argue that such mechanisms are unlikely to remain successful as the *only* mechanism given the expected increase in malicious software for mobile environments. In particular, markets do not have access to all traffic generated by such applications, nor the perfectly analyze all incoming applications prior to distribution.

- **Design and implementation of advanced prototype malware:** We build three proof-of-concept malware instances based on samples collected "in the wild". We then demonstrate the ease with which such malware can resist static and dynamic analysis in a market, making their pre-deployment detection unlikely and the need for a more dynamic "kill switch" mechanism.

- **Cooperatively neutralize malware on infected mobile phones:** We pair a software agent running within a hardened, modified version of the stock Android Linux kernel with a network sensor running Snort. Our network sensor initiates a range of remediation actions that successfully sanitize our advanced prototype malware in under 10 ms. Remediation results can be fed back to mobile markets, allowing both providers and application vendors to more quickly protect customers from malware.

The remainder of this paper is organized as follows: Section 2 compares related research. Section 3 presents an overview of the current state of mobile malware creation and detection, and it demonstrates by example how mobile malware can implement evasive behaviors similar to those of desktop-class malware. Section 4 proposes our remote repair architecture for in-network detection of malware and automatic on-device remediation of the infection. Section 5 presents our specific prototype implementation for Android and an evaluation of the prototype's performance using the evasive malware instances first presented in Section 3. Section 6 includes a discussion of the system, its potential applications, and its current limitations. We conclude in Section 7.

2. RELATED WORK

Mobile malware is becoming increasingly commonplace. While malicious software for these platforms is not new [17–20], the migration towards a small number of mobile operating systems and the increasing power of such devices heightens the probability of large scale infection. Traynor et al. [40] observed that even small numbers of compromised and coordinated phones could be used to cause widespread outages. A number of other authors have built advanced proof-of-concept mobile malware capable of deciphering DTMF tones [36] and collecting video [42]. Mobile malware in the wild has already been found to exfiltrate sensitive data [38], generate calls and text messages to premium numbers [25] and exhibit traditional botnet behaviors [35].

The industrial and research communities have responded with a range of solutions. Numerous anti-virus products have been ported to the most popular mobile platforms [2,8, 16,26]. Others have proposed new platform-based detection systems based on triggers including service thresholds [7], low level API calls [6] and anomalous power usage profiles [27,29]. These device-only approaches fail for a number of reasons, including excessive power requirements for traditional anti-virus system scanning and false positives caused by untrained benign behavior. Tools such as TaintDroid [15] and PiOS [14] can identify information flows in applications, but they require significant manual assistance and do not automatically identify "malicious" behavior; rather, they identify the presence of specific information flows (e.g., IMEIs and IMSIs). Neither approach is currently scalable as a means of analyzing all available mobile applications.

Traffic attribution and provenance are generally difficult problems. IP addresses are well known as weak indicators of the origin of malicious traffic on the Internet. Regular backscatter traffic measurements indicate wide-spread address spoofing [31]. Because cellular networks maintain a cryptographic relationship with each user device, attribution of a specific packet to an individual device is possible. However, additional steps must be taken to identify the specific process responsible for traffic. In the context of virtualized desktop systems, Srivastava et al. [39] developed a virtual machine introspection based technique that attributes network traffic to an executing process for the purpose of implementing firewall rules. Our work uses an in-kernel approach to identify a similar correlation between network behaviors and process execution for the purpose of malware infection remediation.

3. MOBILE MALWARE

Malware has begun to move from desktop computing into mobile environments. This section briefly considers mobile malware already deployed, and it then extensively presents highly capable laboratory samples that we created to demonstrate both the evasive technologies available to malware authors on Android platforms and the utility of our proposed remote repair design.

3.1 In the wild

By the first quarter of 2011, over 1000 mobile malware instances have been discovered in the wild, primarily targeting the Symbian OS for feature phones [30]. Android is the second most common target, and the most common smartphone OS victim. Google, the producer of Android and operator of the official app marketplace, has exercised its revocation ability at least three times since 2010 [3,10,11]. The malicious behaviors of Android malware include [30]:

- Privilege escalation to root (DroidDream).
- Bots (Drad.A).
- Data exfiltration (DroidKungFu, SteamyScr.A).
- Backdoor triggered via SMS (Bgyoulu.A)

Similarly, jailbroken devices running Apple iOS experienced a botnet in 2009 [35]. These attacks send and receive traffic across data or voice channels that can be identified by a network intrusion detection system as malicious.

Detection of mobile malware prior to its deployment is as challenging as detection of malware on desktop systems. Centralized app marketplaces provide an opportunity for centralized analysis before the marketplace lists an app. Unfortunately, marketplaces have at least two deficiencies that

will enable the proliferation of malicious apps. First, malware authors can write their apps with logic to evade detection or analysis, as we show below with laboratory malware samples. Second, the Android platform allows users to install apps from third-party marketplaces that may make no efforts to verify the safety of the software that they distribute. As a result, we naturally expect malware to successfully reach client devices.

3.2 Enhanced prototype malware

As seen in desktop security, the success of traditional malware detection diminishes as malware incorporates strategies designed to evade detection or analysis. While evasions strengthen the need for post-infection remediation, existing mobile malware samples have not yet added such complexity. To better illustrate the effectiveness and flexibility of Airmid, we created laboratory malware instances suggestive of future malicious developments in the mobile environment. Each instance combines malicious functionality now occurring in real mobile malware with evasive functionality common to desktop malware, and embeds the malicious logic into a benign app. Inserting malware into a benign app and rehosting the app on a third-party marketplace has become a common malware distribution strategy. For safety, we created our laboratory malware samples without distribution or propagation functionality.

We created three laboratory mobile malware samples: a Twitter client that leaks private data ("Loudmouth"), a Facebook client sync app that dials premium numbers ("2Faced"), and a mobile bot ("Thor"). Each piece of malware is actually a maliciously modified open-source Android application. Our malware *requires no additional permissions* from the unmodified applications, which are listed in Table 1. All samples include evasive techniques and exceed the capabilities of mobile malware currently in the wild.

3.2.1 Loudmouth

Our first malware instance, Loudmouth, combines:

- Malicious mobile functionality: *Data exfiltration.*
- Evasive functionality: *Malware analysis environment detection.*
- Benign host app: *Twitter client.*

The principal malicious functionality of Loudmouth is data exfiltration, a powerful attack in a mobile context. As users embrace the functionality of smartphones, they place significant personal data on their devices. Even basic feature phones store contact lists, messages, and call databases. GPS-enabled devices can leak location information, turning a mobile device into a tracking system. Such data is useful for criminal activities.

Loudmouth appears to the user as a benign Twitter client. We augmented the Nanotweeter [33] codebase with just 143 lines of new, malicious code. The app provides the expected usual functionality, but it also leaks private information to a server owned by the attacker (here, us) during use. It steals Twitter account credentials, the phonebook database, and the most recently available GPS location. When active, Loudmouth creates malicious network traffic that contains private data and is transmitted to the malware author's server.

Desktop-class malware is commonly analyzed by executing the malware within a virtualized or emulated environment. Malware authors have hence begun to include checks for analysis environments, and the malware alters its behavior so that it does not exhibit malicious functionality during analysis [12, 21]. Mobile malware does not yet contain such sophistication; however, should application marketplaces begin to verify apps via runtime analysis, then mobile malware authors may begin to include checks for analysis environments. We implemented multiple checks in Loudmouth.

To thwart dynamic analysis, Loudmouth contains several checks to *determine its environment*, and it hides all malicious behavior when run on an emulator or a developer phone. The Android platform provides straightforward methods of detection. Loudmouth first queries the "brand", "device", and "model" Java environment variables and compares their values against a whitelist of known consumer phones. The malware then checks for the Radio Interface Layer (RIL) library, the Google Maps application, and any DRM system components. These three components are either supplied by hardware device manufacturers or are proprietary, and hence none are distributed with the emulator or with developer phones. These restrictions are actively enforced even for third-party distributions of the operating system [41]. Loudmouth can thus determine whether the phone is emulated or a physical device, as well as specific details of the phone. To the best of our knowledge, we are the first to investigate and implement these techniques in mobile phones.

3.2.2 2Faced

Our second malware instance, 2Faced, combines:

- Malicious mobile functionality: *Premium number dialer.*
- Evasive functionalities: *Log sanitization* and a *hidden native process.*
- Benign host app: *Facebook sync.*

Premium number dialer applications are examples of historic malware that once targeted dial-up Internet users. These attacks resulted in revenue for the owners of the premium numbers, as users had little chance of refuting their expensive phone bills [9]. These attacks have recently seen a resurgence on mobile devices [25].

2Faced is a new generation of the premium number dialer. It is a Facebook contact sync app that dials premium numbers late at night when the mobile device is not in use. The attacker can generate revenue until the user of the device notices these hidden calls. It creates detectable, malicious telecom traffic to the premium numbers.

Our malware instance includes two evasive actions designed to prolong its existence on infected devices. First, it evades early detection from a vigilant user by removing its entries from the device's call logs. Even observant users will be unable to detect malicious activity, maximizing both the financial loss for the user and the gain for the attacker. Second, the actual malicious dialer functionality is hidden in an ARM executable (2facedsrv) installed by 2Faced into the device's local storage. We evade detection via static app analysis by distributing 2facedsrv inside the image resources of 2Faced, though the executable could also be retrieved remotely. The native process is only 14 lines of code, and its

Table 1: Summary of permissions requested by sample mobile malware.

Permissions	Loudmouth	2Faced	Thor
Network access	✓	✓	✓
Coarse location	✓		✓
GPS location	✓		
Read contacts		✓	
Write contacts		✓	
Call phone		✓	

extraction and invocation from 2Faced requires fewer than 100 lines of code in the Java-based portion of 2Faced. This delivery method has recently been observed in the iKee.B iPhone botnet [35].

Native processes are particularly useful to a malware author. Here, 2facedsrv is able to execute its malicious behavior because it is launched with the same permissions as the Java-based parent 2Faced. Notably, it is not terminated when the user exits the parent application, but rather runs until the device reboots. After a reboot, it then restarts when 2Faced is relaunched. Native ARM processes are not listed in Android's default interface, so only the most observant and expert users might detect it by reviewing a full process list.

3.2.3 Thor

Our third and final malware instance, Thor, combines:

- Malicious mobile functionality: *Bot client.*
- Evasive functionality: *Persistence across reboot.*
- Benign host app: *Weather display.*

Our last malware sample demonstrates how current desktop malware can become equally dangerous in mobile environments. We ported to Android an existing Windows-based botnet client [24], which we refer to as Thor (1,353 lines of code). Its original functionality included system command execution, local file access, and denial-of-service (DoS) attacks using HTTP, UDP, and TCP SYN floods. For safety reasons, our port removed its propagation functionality and pointed its command-and-control (C&C) connection to machines under our control. We added the bot client abilities to MyWeather [32], an open-source weather application. When executing, it generates a malicious network footprint containing both C&C and attack traffic.

Unlike the previous malware instances, Thor is installed as a service separate from MyWeather. The service is immediately active and restarts on every boot of the device. It first connects to two different predefined IRC channels used as the main control channel for the botnet. After connecting, Thor waits for one of our commands, including data exfiltration or DoS.

Mobile bot clients can be used for targeted attacks against the cellular network. By using the GPS location, the botnet can target specific base stations or regions for attack. Bot clients with additional capabilities can potentially augment these location-based denial-of-service attacks with both voice and text message traffic, making attacks more effective. In large numbers, such compromised devices can bring down large portions of the cellular network [40].

4. ARCHITECTURE

We expect that malware such as that presented in Section 3 will ultimately be successfully installed on unwitting mobile devices regardless of protective measures taken to prevent their spread. Automated remediation provides opportunities to repair infected systems. Here, we describe the design of our proposed remote repair system, Airmid, which implements the detection, attribution, and remediation of malware infections in mobile environments.

4.1 Threat model & design principles

Airmid detects and responds to malicious software execution on cellular and mobile devices. This is a straightforward threat model encompassing two primary types of attack. First, we expect attackers to successfully install malware on mobile devices via a variety of usual mechanisms, such as drive-by downloads or automated propagation, and by mobile-specific techniques, including distribution on official and third-party app marketplaces. Second, attackers can subvert the correct execution of a benign app by exploiting a security defect in the app's design or implementation. Both attacks result in undesirable software execution on the targeted device, possibly with root access.

In this threat model, the correct functioning of our proposed prototype system then depends on several reasonable assumptions holding true. We assume the existence of the following:

- A protected software layer on the device lower than the level at which the malware executes. Our remediation design includes on-device software that, if subverted, will result in unsuccessful recovery. Reasonable candidate layers include the kernel (if kernel-level malware can be prevented) and a hypervisor (if virtualized environments can be created on a mobile device). For our prototype implementation, we chose to operate at the kernel level and harden the kernel appropriately.
- A communication channel between the network and each device, even if the channel is intermittently connected. Our cooperative design sends signals from network detection systems to software agents on infected systems, and we expect that these signals will ultimately reach any device. While the permanent absence of a communication channel prevents remediation, it likewise prevents malice from spreading beyond the device.
- Detectable malicious behavior in the network. We chose to use network-based attack detection rather than on-device detection to avoid the costly process of traditional signature-based malware detection. In mobile environments, the storage, computation, and power costs of malware detection may be prohibitive, and moving detection into the network avoids those costs altogether.

We have no ability to prevent device owners from rooting or jailbreaking their devices and then overwriting the

Airmid software. Reflashed devices simply will not benefit from automated repair of infections and may simply be denied service by their provider should malicious traffic be detected (as currently happens).

4.2 Remote repair

Our remote repair design consists of a server-based infection detection system and an on-device attribution and remediation system that cooperate to automatically identify and repair infected devices. The network-based detection system has a broad view of a provider's entire network: it can observe behaviors correlated across multiple devices that would not be meaningful to any single device, such as a distributed denial-of-service attack or exfiltrated data all transmitted to the same server. The on-device software permits detailed, real-time inspection of device state and the ability to effect changes that restore benign operation. The server automatically initiates remediation of infected devices via an authenticated communication channel. Airmid's cooperative design allows these strengths to be combined: broad network perspective, detailed device knowledge, and effective attack response.

During an active infection, Airmid operates as shown in Figure 1. Depending upon the nature of the specific infection, an infected device sends and receives malicious data—such as propagation traffic, command-and-control, and exfiltrated data—and generates calls or text messages to high-cost recipients. A network sensor (which could be using data sources including DNS blacklists, patterns of known botnets [22] or other technique) monitoring data and telecom networks (1) identifies infected devices based upon these malicious traffic patterns. It notifies (2) the Airmid server of the offending devices and traffic, and the server sends the alerts across the authenticated channel to each infected device (3). The on-device software receives the alert (4), inspects the kernel's internal state to attribute the malicious network traffic to a particular process or service, and then initiates one or more remediation actions against that process.

The Airmid on-device software executes from within a trusted, low layer on the device, such as a hardened kernel or hypervisor. It has two responsibilities: attribute traffic to malware, and repair the infection. Attribution is closely tied to a particular mobile operating system's implementation, so we defer the presentation of our Android-specific analysis to Section 4. Once it has identified software on a device responsible for whatever network behaviors triggered the detection system, the software executes repair actions to disable malicious activity or to remove malware entirely. We employ the following recovery options, loosely ordered from least likely to create side-effects to most likely:

- Process termination.
- On-device traffic filtering.
- App update.
- Device update.
- File removal.
- Factory reset.

Decision logic within the software component will select an action based upon the nature of the offending network traffic, the identity of the attributed process, and that process' privilege level.

4.3 Authenticated communication

The Airmid software executes attribution and recovery when signaled by the network intrusion detection system. The communication channel carrying these signals clearly must be secured to prevent illicit triggering of operations that change a device's state. Airmid authentication augments initial UMTS authentication in a cellular network, as shown in Figure 2. It extends the scope of authentication from the device and network to the Airmid device and server components, reusing existing network functionality and secret keys. This has three benefits: first, no additional secrets need to be shared or updated for this authentication, as one private key manages both authentication types. Second, the difficulty of integration into existing networks is reduced, as no additional authentication servers are required. Third, the approach provides the same guarantees as existing UMTS authentication and can be upgraded at the same time if needed.

The final step of UMTS authentication completes the mutual authentication and sets a session key used for encryption. Airmid extends this authentication process.[2] All traffic is encrypted through the existing session key, which guarantees a secure connection through the cellular network. At this point the Mobile Switching Center (MSC) notifies the Airmid server of the newly authenticated device. The server then requests UMTS Authentication Vectors (AV) from the Home Location Register (HLR), the device storing the long-term credentials for traditional telephony authentication. These AVs consist of the RAND, XRES and AUTN values, which are the randomly generated user challenge, expected user response, and the authentication token respectively. The Airmid server then forwards its values of RAND and AUTN to the device. Through these values the device component verifies the server's knowledge of the secret key and generates its response based on its key. This response allows the Airmid server to verify that the device is aware of the secret key, hence, completing mutual authentication. If the server or device are unable to prove their knowledge of the shared secret, then the authentication fails. The session key $K(a)$, which is never sent over the air, is then used for encryption. Cryptographic algorithms used in UMTS are similarly used for Airmid authentication and encryption.

5. IMPLEMENTATION

Our prototype implementation is separated into two components: passive network analysis and signaling within the network, and remediation logic that responds to these signals on the device. To emulate sensors within the cellular network, we built a packet sniffer that analyzes network traffic from an HTC Dream mobile phone running Android 1.6. A secure communication channel is established between the sensor and the kernel when a sensor needs to issue remediation commands to an infected device, whereupon the device performs various actions to remediate the infection.

[2]We encourage readers interested in more information about UMTS authentication procedures to read either the standards document [1] or Traynor et al. [40].

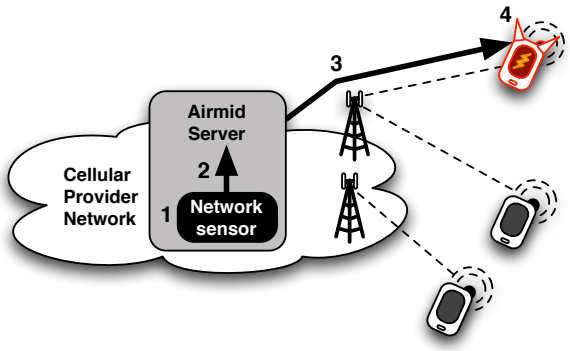

Figure 1: High-level workflow of automated remote repair. (1) An infected device sends or receives malicious data or voice communication detected by network sensors deployed within a provider's network. (2) The provider notifies an Airmid server of the offending traffic. (3) The server transmits an authenticated message to the protected client-side Airmid software on the infected device. (4) The software attributes the malicious network traffic to a responsible process, and initiates remediation actions.

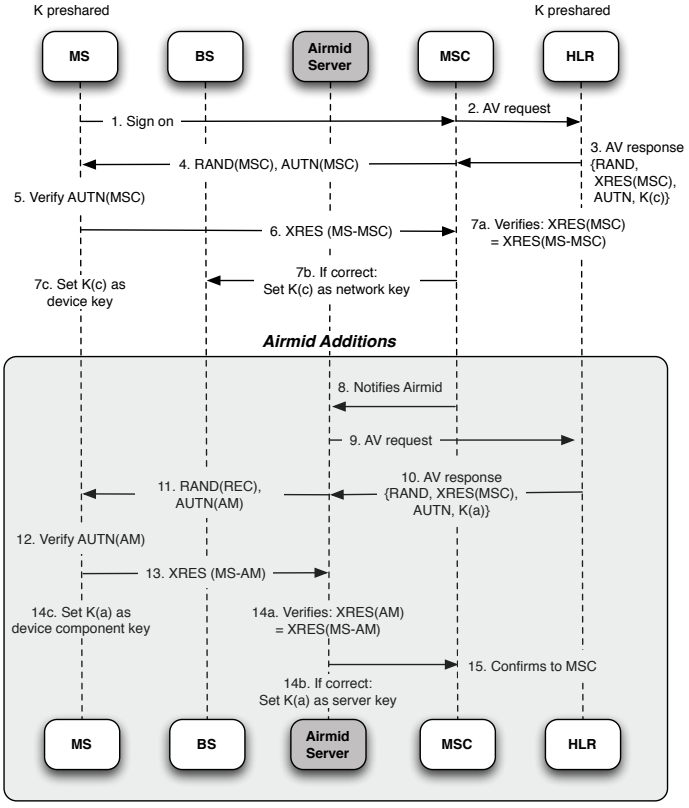

Figure 2: Authentication flow: the Airmid authentication extends existing UMTS authentication. The device and network are mutually authenticated after UMTS authentication and all further traffic is encrypted using a session key. Airmid extends this authentication process to create a secure channel between the Airmid server and the process running on the mobile phone.

5.1 Network component

Our prototype network component includes both an attack detection system and the Airmid server communicating with infected devices. For detection, we used Snort, an off-the-shelf network intrusion detection system. We wrote our Airmid server in Python using the packet creation library Scapy [5]. As a mobile device communicates with foreign network assets, Snort analyzes incoming and outgoing traf-

fic. When it flags traffic as malicious, it notifies the Airmid server which subsequently initiates authenticated two-way communication with the suspect device. It sends to the device a remediation message that provides the device's port number from the attack alert. This information is used by the Airmid on-device software to determine which processes and applications are acting maliciously. The device's software decides which remediation strategies are appropriate and relays how the infection was remediated to the sensor.

5.2 Device component

Our threat model considers a device's kernel to be the trusted computing base, so provenance and remediation actions run in kernel-mode. We modified the Linux kernel version 2.6.29 to include our prototype implementation. We hardened the kernel against direct compromise by disabling its ability to dynamically load kernel modules. Our modifications required approximately 1,200 lines of C. When the device is booted, a separate kernel thread is launched to perform infection provenance and remediation. When a message is received from an authenticated Airmid server, the device: determines provenance of the offending traffic, determines and enacts a remediation strategy, and relays the device's decision to the network sensor. We now discuss the implementation of the provenance mechanism that leverages Android's existing application sandboxing techniques as well as implementation details for the remediation strategies.

5.2.1 Infection provenance

When a user installs an application on a mobile device, Android generates a unique Linux user ID that persists for the application's lifetime [4]. All files owned by the application are also assigned this user ID. Given a user ID, we can find the Android package file (`.apk`) of the application as well as configuration files and native binaries dropped by the application both at install-time and runtime.

When a device receives a remediation signal from a network sensor, the device crawls two kernel data structures: `inet_hashinfo`, which contains port usage information, and `task_struct`, which contains process information. While `inet_hashinfo` contains port information only for TCP connections, the technique itself is general and could be extended to perform provenance on UDP-based traffic. The port number sent to the device by the network sensor is mapped to an open socket by iterating through `inet_hashinfo`. We then look for this socket in the list of `task_struct`s to find the process that holds said socket. A visualization of this process is shown in Figure 4. From the `task_struct`, we then extract the user ID, u, of the malicious process and crawl the disk to find all files owned by this application.

5.2.2 Remediation strategies

Depending on the value of u, the user ID identified in the previous step, the Airmid software initiates one or more of the following repair actions: creation of kernel-protected local firewall rules to block the malicious traffic, termination of processes running under u, removal of the application package (apk) owned by u, and removal of all files owned by u. If $u < 10000$, then it is a system user ID corresponding to a core service whose benign functionality may have been subverted. Airmid will only block the malicious traffic by creating appropriate firewall rules. If $u \geq 10000$, then u is an application user ID. Airmid will terminate all processes running with ID u and will remove the application package owned by u. We identify applications for removal by parsing the file containing all installed application packages and the user IDs given to them Android at install-time (`/data/system/packages.xml`). Finally, Airmid scans the list of running processes to see if any native ARM processes are executing with user ID u. If so, it scans the full storage of the device to purge all files owned by u.

To firewall malicious traffic, the client interfaces with the Android kernel's netfilter [34] hooks to provide lightweight packet filtering. We add rules that prohibit traffic destined for the IP addresses actively in use by processes owned by u. Additional provenance checks determine all active IP addresses if more than one process is running under u. To prevent observation or alteration of rules from userland, Airmid does not register its rules with `ip_tables` but rather maintains its own shadow `ip_tables` data structure.

Table 2: Average Command Execution Time (95% confidence interval)

Command	Mean (μs)	C.I. (μs)
Factory Reset	1504	± 38
Device Status (Process List)	2991	± 48
Process Termination	6537	± 1423
File Removal	6115	± 31
Device-Side Filtering Rule	7149	± 826
Application Removal	9174	± 1123

5.3 Performance evaluation

We characterize the overheads associated with the Airmid architecture through a performance analysis. We measured each operation 100 times and provide 95% confidence intervals.

We measured each of the operations discussed in Section 4.2 and recorded very modest overheads for all proposed functions. Table 2 summarizes our experiments. While seemingly surprising, the factory reset function takes the least time with an average of 1504 μs ($\pm 38\mu$s). However, a factory reset can be performed simply by deleting a single configuration file (`data.img`). The most expensive operation, application removal, requires the most time with an average of 9174 μs ($\pm 1123\mu$s). This result was also somewhat expected as it includes the costs of both process termination and deletion of multiple files.

These results demonstrate the lightweight nature of Airmid. Unlike more traditional anti-virus architectures, which require regular scanning of all of the contents of the phone (an operation taking on the order of 10s of seconds), our approach attributes malicious behavior directly to an application and allows a targeted response to be implemented. This makes our approach more conscious of the power constraints associated with mobile devices.

6. DISCUSSION

The concept of automating remote infection identification and local device repair raises natural questions over the organizational control of remediation and the security of the required on-device software.

6.1 Airmid control

Airmid provides a powerful architecture to respond to the growing problem of mobile malware. While we believe that this approach can help the vast majority of individual users,

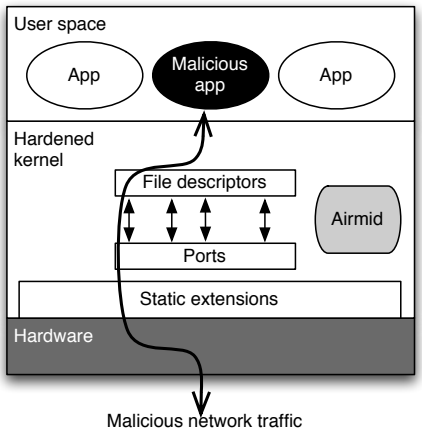

Figure 3: Airmid's on-device component resides within a hardened Android kernel. In particular, all kernel extensions are statically linked into the kernel so that its loadable module support can be disabled.

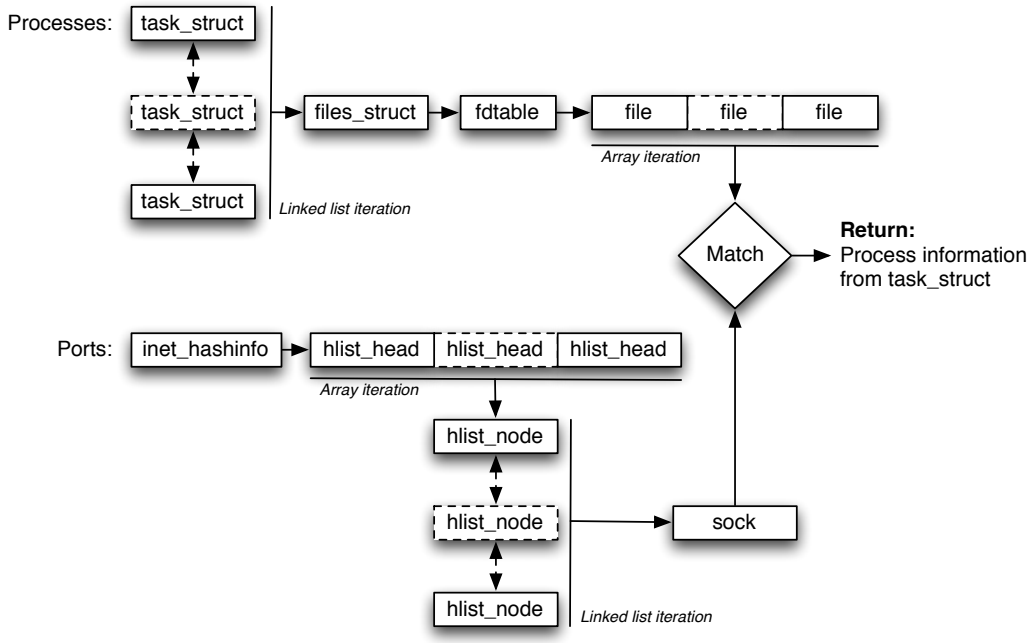

Figure 4: Traversal of data structures to perform malicious traffic provenance of the mobile device.

we recognize that there exist numerous parties that may not trust a cellular network provider to perform these operations. For instance, phones used by members of government agencies or employees of a rival company may not wish to outsource their malware remediation. *We do not intend Airmid to be a "one size fits all" solution* and discuss how such systems can be deployed in reality.

Our architecture is general enough such that the functions described in this paper can be implemented by a separate entity or cloud service. Many companies already require data traffic from their corporate phones to be proxied via VPN through their network.[3] Such entities could simply deploy an Airmid-speaking service in their network and then configure their devices (e.g., by changing keying material on the device) to only accept commands from that server. Individuals not wanting such control could similarly alert the provider, perhaps through an out-of-band resource such as a web interface or when the device/service plan is purchased. We leave the debate over opt-in/opt-out strategies to the policy community, but note that a range of economic incentives could be provided to sway customers towards the use of Airmid regardless of the chosen approach.

[3]We note that the Blackberry BES provides VPN service and offers to limit the applications allowed to run on mobile devices. Our architecture is more flexible and does not derive its protections from the explicit specification of applications.

Airmid does not tie a device's security to arbitrary cellular providers. For instance, when traveling abroad, mobile phones running the modified Airmid kernel would be able to receive roaming telephony and data service as normal without providing their "visited" network with control over Airmid's functionality. We believe that this approach is necessary as laws potentially protecting customers from abuse by their provider domestically are unlikely to apply internationally. While this means that misbehaving roaming devices are more likely to simply be denied service by their visited network (as is currently the practice), we believe that this tradeoff is ultimately more secure.

Lastly, the guarantees provided by Airmid are only as good as the detection mechanisms used by the network sensor. For instance, policies enacted against traffic from or destined for known malicious domains or bots are likely to be effective. However, a system relying on an IDS reporting any "anomalous" traffic will produce false positives. Accordingly, these mechanisms must be selected carefully. We leave this selection to future work, where we intend to observe and further characterize the network traffic of large amounts of mobile malware.

6.2 Device hardening

The correct operation of our architecture requires that the device is able to protect the Airmid software. We chose to do this in our proof of concept implementation by hardening the stock Android Linux kernel through steps including disabling dynamic loading of kernel modules. Given that a significant proportion of traditional malware abuses this mechanism and that mobile devices generally do not take advantage of this capability, we believe that this is a first strong step in preventing kernel compromise. However, we recognize that deploying Airmid on real systems may require additional hardening. An increasingly popular approach is the use of virtualization. While a number of virtualization solutions are evolving for mobile devices [28], we believe that such mechanisms are relatively expensive and may themselves not yet be ready for widespread deployment. As our threat model indicates, it is necessary to identify a protection layer with reasonable tradeoffs in which Airmid can run. We believe that our approach is reasonable given current mechanisms, but note that best practices for device hardening are still an active research area.

7. CONCLUSION

As mobile devices begin to see an increasing volume of malicious applications, the ability of application markets to identify and remove such applications in a timely manner will be lost. We respond to this problem by developing Airmid, an automated system for the remote remediation of mobile malware. Upon the detection of malicious traffic, the cellular network interacts directly with the source device to identify the provenance of that traffic. The device can then perform remediation ranging from filtering the offending traffic to uninstalling the application. We then demonstrate that Airmid has very low overhead. In so doing, we demonstrate that the detection and removal of malicious applications can scalably be outsourced to cellular providers and applications markets, ultimately providing faster responses to infection.

Acknowledgments

We would like to thank Michael Iannacone, Ferdinand Schober and our anonymous reviewers for their helpful comments in the completion of this paper. This work was supported in part by the US National Science Foundation (CNS-0916047). Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] 3rd Generation Partnership Project. General packet radio service (GPRS). Technical Report 3GPP TS 23.060 v8.0.0.

[2] Airscanner AntiVirus for Windows Mobile, Accessed 2011. `http://www.airscanner.com/downloads/av/av.html`.

[3] C. Albanesius. Google pulls malware-infected apps from Android market. *PCmag.com*, June 2011.

[4] Android Developers. Security and permissions, Accessed 2011. `http://developer.android.com/guide/topics/security/security.html`.

[5] P. Biondi. Scapy. `http://www.secdev.org/projects/scapy/`, Accessed 2011.

[6] 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)*, June 2008.

[7] A. Bose and K. Shin. Proactive security for mobile messaging networks. In *Proceedings of the ACM Workshop on Wireless Security (WiSe)*, Sept. 2006.

[8] Bullguard mobile antivirus, Accessed 2011. `http://www.bullguard.com/why/bullguard-mobile-antivirus.aspx`.

[9] R. E. Calem. Scam costs net users thousands in transatlantic telephone bills. *New York Times*, Feb. 11, 1997.

[10] R. Cannings. Exercising our remote application removal feature. Android developers blog, June 2010. `http://android-developers.blogspot.com/2010/06/exercising-our-remote-application.html`.

[11] R. Cannings. An update on Android market security. Google Mobile blog, Mar. 2011. `http://googlemobile.blogspot.com/2011/03/update-on-android-market-security.html`.

[12] X. Chen, J. Andersen, Z. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, June 2008.

[13] D. Dagon, T. Martin, and T. Starner. Mobile phones as computing devices: The viruses are coming! *IEEE Pervasive Computing*, 3(4):11–15, 2004.

[14] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *Proceedings of the ISOC Networking & Distributed Systems Security Symposium (NDSS)*, Feb. 2011.

[15] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2010.

[16] F-Secure mobile security, Accessed 2011. `http://www.f-secure.com/en_US/products/mobile/mobile-security/`.

[17] F-Secure Corporation. F-Secure computer virus descriptions: Cabir, Dec. 2004. `http://www.f-secure.com/v-descs/cabir.shtml`.

[18] F-Secure Corporation. F-Secure computer virus descriptions: Mabir.A, Apr. 2005. `http://www.f-secure.com/v-descs/mabir.shtml`.

[19] F-Secure Corporation. F-Secure computer virus descriptions: Skulls.A, Jan. 2005. http://www.f-secure.com/v-descs/skulls.shtml.

[20] F-Secure Corporation. F-Secure malware information pages: Worm:SymbOS/Commwarrior, 2008. http://www.f-secure.com/v-descs/commwarrior.shtml.

[21] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is not transparency: VMM detection myths and realities. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, May 2007.

[22] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. BotHunter: Detecting malware infection through IDS-driven dialog correlation. In *Proceedings of the 16th USENIX Security Symposium*, Aug. 2007.

[23] C. Guo, H. J. Wang, and W. Zhu. Smart phone attacks and defenses. In *Proceedings of Third ACM Workshop on Hot Topics in Networks (HotNets)*, Nov. 2004.

[24] Hack Forums. Windows botnet source, Accessed 2011. http://www.hackforums.net/showthread.php?tid=108411.

[25] InfoSecurity.com. Premium rate calling Android malware spotted in the wild, May 2011. http://www.infosecurity-us.com/view/18301/premium-rate-calling-android-malware-spotted-in-the-wild/.

[26] Kaspersky mobile security, Accessed 2011. http://www.kaspersky.com/mobile_downloads.

[27] H. Kim, J. Smith, and K. Shin. Detecting energy-greedy anomalies and mobile malware variants. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2008.

[28] L4Android - Android on top of L4, Accessed 2011. http://l4android.org/.

[29] L. Liu, G. Yan, X. Zhang, and S. Chen. Virusmeter: Preventing your cellphone from spies (sic). In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2009.

[30] McAfee Labs. McAfee threats report: First quarter 2011, June 2011.

[31] D. Moore, G. M. Voelker, and S. Savage. Inferring Internet denial-of-service activity. In *Proceedings of the USENIX Security Symposium*, Aug. 2001.

[32] My Weather, Accessed 2011. http://island.byu.edu/unclass/content/android-web-service-app-my-weather-running-and-full-source-code.

[33] Nanotweeter, Accessed 2011. http://code.google.com/p/nanotweeter/.

[34] Netfilter/iptables project, Accessed 2011. http://netfilter.org/.

[35] P. Porras, H. Saidi, and V. Yegneswaran. An analysis of the iKee.B (Duh) iPhone botnet. Technical report, SRI International, Dec. 2009.

[36] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *Proceedings of the ISOC Network and Distributed Systems Security (NDSS) Symposium*, Feb. 2011.

[37] SecurityWeek News. New malware jumps to 73,000 samples every day, says PandaLabs, Mar. 2011. http://www.securityweek.com/new-malware-jumps-73000-samples-every-day-says-pandalabs.

[38] M. Shipman. Enter the hacker: New DroidKungFu malware is bad news for Androids. The Abstract blog, June 2011. http://web.ncsu.edu/abstract/technology/wms-droidkungfu/.

[39] A. Srivastava and J. Giffin. Tamper-resistant, application-aware blocking of malicious network flows. In *Recent Advances in Intrusion Detection (RAID)*, Cambridge, Massachusetts, Sept. 2008.

[40] P. Traynor, M. Lin, M. Ongtang, V. Rao, T. Jaeger, T. La Porta, and P. McDaniel. On cellular botnets: Measuring the impact of malicious devices on a cellular network core. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Nov. 2009.

[41] T. Wimberly. Cyanogenmod in trouble? Android and Me blog, Sept. 2009. http://androidandme.com/2009/09/hacks/cyanogenmod-in-trouble/.

[42] N. Xu, F. Zhang, Y. Luo, W. Jia, D. Xuan, and J. Teng. Stealthy video capturer: A new video-based spyware in 3G smartphones. In *Proceedings of the ACM Conference on Wireless Network Security (WiSec)*, Mar. 2009.