Security Analysis of Consumer-Grade Anti-Theft Solutions Provided by Android Mobile Anti-Virus Apps

Laurent Simon
University of Cambridge
lmrs2@cam.ac.uk

Ross Anderson
University of Cambridge
rja14@cam.ac.uk

Abstract

We study the "anti-theft" mechanisms available to consumers to thwart unauthorised access to personal data on stolen Android smartphones. With millions of devices stolen in the USA in 2013 alone, such attacks are a serious and growing problem. The main mitigation against unauthorised data access on stolen devices is provided by "anti-theft" apps; that is, with "remote wipe" and "remote lock" functions. We study the top 10 Mobile Anti-Virus (MAV) apps that implement these functions. They have been downloaded hundreds of millions of times.

We investigate the general security practices of MAVs, as well as the implementation of their "remote wipe" and "remote lock" functions. Our analysis uncovers flaws that undermine MAV security claims and highlight the fragility of third-party security apps. We find that MAV remote locks may be unreliable due to poor implementation practices, Android API limitations and vendor customisations. Mobile OS architectures leave third-party security apps little leeway to improve built-in Factory Resets, therefore MAV remote wipe functions are not an alternative to a flawed built-in Factory Reset. We conclude the only viable solutions are those driven by vendors themselves.

I. Introduction

The extraction of personal data on stolen devices is a growing concern. In 2012, smartphone robberies were almost 50% of all robberies in San Francisco, 40% in New York City and were up 27% in Los Angeles[1]. In 2013, 3.1M devices were stolen in the USA[2], and 120,000 in London[3]. For the half of all users who do not lock their screen[4], the main anti-theft protections in use today are "remote wipe" and "remote lock" functions. Products that offer these remote anti-theft data protections include a range of enterprise and consumer-grade offerings. In this paper, we focus on the latter and study the top 10 Mobile Anti-Virus (MAV) apps. The Google Play store indicates that these apps are prevalent today with the top 2 MAVs downloaded between 100M and 500M times each, the third between 50M and 100M times, and the following 4 between 10M and 50M. In comparison, the top enterprise app (for mobile device management) was downloaded less than 5M times. Our analysis reveals eight issues: (*i*) misinformation given to users following a remote wipe and lock; (*ii*) questionable MAV authentication practices; (*iii*) the limitation of, and the restriction imposed by, Android's APIs and architecture; (*iv*) the misuse of Android security APIs by MAVs; (*v*) inconsistency of Android's API across versions; (*vi*) incorrect Android API documentation; (*vii*) MAV reliance on carrier network (in)security; and (*viii*) unfortunate customisations by vendors. For example, we found that because of Android API differences across versions, 9 MAVs can be uninstalled by a thief before a Gingerbread (v2.3.x) phone is remotely locked by its user; because of API misuse, 4 MAV locks can be bypassed (Section VI); and because of vendor customisations, all MAV locks can be circumvented. By comparing MAV remote wipe functions with previous findings on Android built-in Factory Resets [1], we find that mobile OS architectural decisions that were aimed at enhancing security (e.g. the permission system and lack of root access) get in the way of MAVs that attempt to improve the reliability of flawed factory resets. Therefore MAV remote wipe functions are not an alternative to a flawed built-in factory reset (Section VII).

In summary, our contributions are as follows:

- We present the first comprehensive study of Mobile AV (MAV) implementations in the context of device theft, including their general security practices and anti-theft functions.
- We uncover major failures that may affect millions of users. These flaws are not only caused by questionable practices by MAV developers, but also by vendor customisations and by the limitation of the Android OS. Remote lock functions can therefore be bypassed, and remote wipe functions are not an alternative to flawed Factory Resets.
- We discuss possible countermeasures, but conclude that only vendor-provided software has the potential to raise the reliability of anti-theft mechanisms.

II. Background

A. Data Partitions

Anti-theft solutions must protect personal data on all storage on the device. Android smartphones share three common partitions for storing a user's data (Fig. 1).

[1] gizmodo.com/5953494/hold-on-tight-smartphone-mugging-is-more-popular-than-ever

[2] www.consumerreports.org/cro/news/2014/04/smart-phone-thefts-rose-to-3-1-million-last-year/index.htm

[3] www.london.gov.uk/media/mayor-press-releases/2013/07/mayor-challenges-phone-manufacturers-to-help-tackle-smartphone

[4] www.consumerreports.org/cro/news/2014/04/smart-phone-thefts-rose-to-3-1-million-last-year/index.htm

The first is the data partition that hosts apps' private directories; it is generally mounted on */data/*. An app's private directory cannot be read or written to by other apps, so it is commonly used to store sensitive information such as login credentials. On older phones with a small data partition, one can also install apps on an external SD card; but this is usually not the default behaviour.

The second partition storing user data is the internal (primary) SD card. Despite its name, it is not an SD card per-se, but a partition physically stored on the same chip. The internal SD card is mainly used to store multimedia files made with the camera and microphone; it is generally exposed to a computer connected via USB – via Mass Storage, Media Transfer Protocol (MTP) or Picture Transfer Protocol (PTP).

The last partition containing user data is the external, removable SD card. It offers similar functionality to the internal SD card, but can be physically inserted and removed by the user. If there is no internal SD card on the device, the external one becomes the "primary SD card"; otherwise it is called the "secondary SD card" (in this case the "primary SD card" is the internal one). The primary and secondary SD cards are sometimes referred to as "external storage".

Some devices also have hardware key storage. When supported, it is used principally by the default Account Manager app.

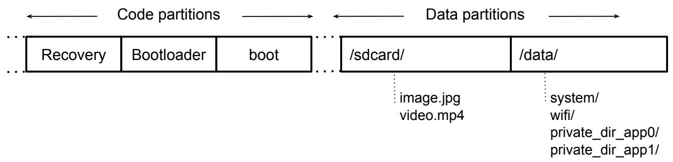

Fig. 1. Common Android partitions. Each rectangle represents a partition on the same flash storage.

B. Level of Sanitisation & Factory Resets

In Section VII, we compare the reliability of Android built-in Factory Resets with the remote wipe functions of MAVs. For comparison, we need to agree on a level of sanitisation. The following three levels of data sanitisation exist [2].

The highest level of sanitisation is *analog sanitisation*, this degrades the analog signal that encodes information, so that its reconstruction is impossible even with the most advanced sensing equipment and expertise.

The second level is *digital sanitisation*. Data in digitally sanitised storage cannot be recovered via any digital means, including the bypass or compromise of the device's controller or firmware, or via undocumented drive commands.

The third level is *logical sanitisation*. Data in logically sanitised storage cannot be recovered via standard hardware interfaces like standard eMMC commands.

Simon and Anderson [1] studied the reliability of built-in Factory Resets in the context of *logical sanitisation* because it is the only one that is cheap and may be profitable if exploited at scale. The authors found that certain phones do not logically sanitise the data partition, the internal SD card and/or the external one. This may affect devices running Android from Froyo (v2.2.x) to KitKat (v4.4).

For a fair comparison, we take the same approach as Simon and Anderson, in that we consider a remote wipe to be "secure" or "proper" if it provides logical sanitisation.

C. Bootloader, Recovery and Safe Modes

Besides partitions storing personal user data, phones also store (binary) executable files in dedicated partitions (Fig. 1). These contain binaries to boot in normal mode (i.e. Android) and other less-known special modes of operation. Android smartphones generally have three extra modes of operation that are useful to our discussion: the Bootloader mode, the Recovery mode and the Safe mode. A user can boot into them by pressing a combination of hardware keys on the device. There exist subtle variations between vendors, so we try to keep the description general.

The Recovery mode is generally a headless Android OS used for performing updates and backups to the current installation: updates may be stored in external storage or sent in-band from a computer connected via USB. The Bootloader is not based on Android, and it allows flashing new software and partitions to a device, generally via USB. To achieve their functionalities, both the Recovery and Bootloader mode must run with high privileges. Typically, this means unrestricted access to both the Android OS binaries and partitions storing user data. There are three kinds of Bootloader and Recovery protections we have found on devices: open, protected, and locked.

Open Bootloaders/Recoveries let anyone with physical access to a device install custom updates. We found this to be true for most Samsung and LG devices in our sample.

Locked Bootloaders attempt to lock devices to a certain carrier or vendor by enforcing signature verification on software updates. This is true of most HTC devices we encountered. To disable the signature verification, a locked bootloader needs to be "unlocked". This may be possible via OS or bootloader exploits. HTC also lets users "unlock" their Bootloader through their website, but voids the warranty of the device thereafter. Upon unlock, the Bootloader is supposed to wipe all data on the device, so as to prevent thieves from recovering users' data after installing forensics software. We note that a locked Bootloader/Recovery is not a panacea: as the key used to sign a software update is owned by the vendor, an insider – or a server or CA compromise – could leak it to attackers. We stress that this is not a hypothetical scenario: for one phone in our sample, we found an implementation of Recovery that passed the signature verification and let us root the device. In practice, a locked Bootloader/Recovery may provide enough security for average users, but not for firms with high assurance requirements.

Protected Bootloaders/Recoveries genuinely try to protect users: the lock does not serve any business purpose. Users are empowered to unlock their Bootloader to install custom software without voiding the warranty. This is mostly true of Google phones. Unlike open and locked Bootloaders, a protected one can be "re-locked". If a thief wants to install forensic software on the device, he can unlock the Bootloader but this will also wipe the device's data.

The Safe Mode boots the main Android OS, but disables all user-installed apps. This is primarily used for users to un-install misbehaving apps, for example malware that may lock the screen and render the phone unusable. Even though apps are disabled in the user interface, they can still be launched via a shell. Obtaining a shell can be achieved by first enabling the Android Debug Bridge (adb) developer option in the default phone Settings; and then plugging the device into a computer via USB. By design, this gives a shell prompt on the computer to interact with the device.

D. Mobile Anti-Virus (MAV) Apps and Device Admin API

Mobile Anti-Virus (MAV) apps have already been downloaded hundreds of millions of times from the Google Play store. They generally achieve their remote anti-theft protections with an app installed on the device in combination with an online web interface accessible from a standard web browser. A user who loses his phone can log in the web interface and remotely instruct the phone app to wipe or lock the device. An exception amongst the apps we studied was *Dr.web*: instead of using a web interface, it requires users to define trusted phone numbers, from which a user can send remote commands to his lost phone via SMS.

A simple attack against anti-theft solutions is the use of "Faraday bags" to block all radio-frequency communications between a stolen device and its cloud service, thereby preventing any remote action from a device's owner. We leave this problem aside for the moment and discuss possible countermeasures in Section VIII. In the following sections, we highlight other important issues which we believe are relevant to improve the reliability of current anti-theft solutions in general.

All MAVs make use of a special set of functions accessible via Android's "Device Administration API", that provides administration features at the system level. Once an application is granted access to this API, it becomes a "device admin" and gains access to security "policies" like the password policy (e.g. to enforce password strength), or the encryption policy. Each policy within the admin set must be explicitly requested in an app's manifest. Fig. 2 shows the relevant code for requesting access to the `force-lock`, `wipe-data`, `reset-password` and `disable-camera` policies. Unlike traditional Android permissions, the admin permission and policies are not granted at installation time: they must be approved all at once by a user in the Android default Settings. When not granted, an app can still run, but without admin privileges. The two admin policies relevant to our study are the wipe and screen lock policies that can be used to protect users' data when devices are lost. At runtime, an admin app with the `wipe-data` policy can invoke the *wipeData(int flag)* function to perform a wipe. It currently supports wiping the data partition only (*flag* $= 0$) or with the additional wiping of the primary SD card (*flag* =*WIPE_EXTERNAL_STORAGE*). The API does not support wiping the secondary (external) SD card. Internally, *wipeData()* uses the device's built-in Factory Reset – so its reliability varies across devices as detailed in Section II-B. An admin app with the `force-lock` policy can also use the built-in PIN screen to lock the phone screen (e.g. by invoking the *lockNow()* function). One additional security protection is that an admin MAV cannot be un-installed unless its admin privileges are first removed in the default Android Settings. Nevertheless, even an admin MAV has limitations: it cannot access other apps' private directories in the data partition, nor can it bypass the file system to read/write arbitrary content from/to storage.

If a user forgets to enable the admin permission for a MAV, the app can neither use built-in wipe and lock features, nor overwrite partitions reliably bit-by-bit to sanitise data storage. Therefore, it must resort to less reliable ad-hoc mechanisms. For example, it may use public Android APIs with the traditional permissions granted at installation time (like contact APIs to remove contacts from the Phonebook app). However, this generally results in the deletion of records in the associated SQLite file, which does not provide logical sanitisation. For external storage, a non-admin app may fill existing files with random bytes, *unlink* them, create new ones (in the hope of overwriting unallocated file system space), or format the partition. User-installed apps generally do not expose sanitisation APIs on the phone, so their data would typically remain intact. As for the screen lock, a MAV could detect when it loses screen focus, and subsequently launch one of its "views" (a.k.a. Android Activities) to foreground.

```
# =========== AndroidManifest.xml ==========
<receiver android:permission="BIND_DEVICE_ADMIN">
  <intent-filter>
    <action android:name="DEVICE_ADMIN_ENABLED" />
  </intent-filter>
  <intent-filter>
   <action android:name="DEVICE_DISABLE_REQUESTED"/>
  </intent-filter>
</receiver>

# =========== device_admin.xml ==========
<device-admin [...] >
  <uses-policies>
    <reset-password />
    <force-lock />
    <wipe-data />
    <disable-camera />
  </uses-policies>
</device-admin>
```

Fig. 2. Device admin request example. Permission and action names are purposely shortened for readability. Two broadcast receivers are declared: one to receive a notification when the user has accepted the admin permission, another when the user is trying to disable it in the default Android Settings. Four policies are requested.

III. Methodology

We restricted ourselves to the 10 most-downloaded MAVs on Google Play – the Google Device Admin app was not in the top 10. We downloaded them from a Samsung Galaxy S Plus phone between Nov. 2013 and Apr. 2014. Our Samsung Galaxy S Plus runs Gingerbread (v2.3.5), has a primary SD card formatted in FAT, and we inserted a secondary 2GB removable SD card in its slot. We conducted a review of apps' code using *apktool*[5] and simple run-time analysis to confirm our findings. We report our findings on the general security of MAV solutions in the following sections, and specifically focus on anti-theft functions in Section VI and Section VII. We also report on the discussions we had with MAV developers after the responsible disclosure of our findings.

IV. Account Authentication

MAVs are sensitive-permission hungry, and their web interface is a proxy to the rich functionalities they offer. Through the web interface, MAVs offer sensitive functions such as access to personal data backups, remotely taking pictures, remotely enabling the microphone, etc. By gaining access to a victim's account, an attacker could therefore remotely access personal information or lock the device to demand a ransom. To protect user accounts, all MAVs in our sample ask users to select a password at first run of the app.

Findings: We found *questionable authentication practices* (Table I, column "Account Password"). For example, all MAVs accept short passwords – ranging from 4 (*Dr.web*) to 8 minimum characters (*Kaspersky*). Four of them do not accept special characters (*McAfee*, *Avira*, *TrendMicro* and *TrustGo*). *Avast* does accept special characters, but processes them somehow: when entering the password *hello"'@#%&*/-+()*, we could then log in with the truncated version *hello"'@#%*. More generally, all apps accept weak passwords, except *Kaspersky* that forced us to use a combination of uppercase, lowercase and numerals.

We hypothesize that in practice, it is difficult for MAVs to enforce strong passwords: as users rarely interact with MAVs or their corresponding web interface, they would inevitably forget their password if not easily memorable. Furthermore, if a password is set up on a phone, the keyboard limitations make it inconvenient to mix upper and lower case, let alone adding non-alpha characters.

We also found indications that certain web services may not store user credentials properly. When websites enforce a maximum password length, it is often indicative of bad storage practices – when stored hashed, passwords can be arbitrarily long [3]. Three MAVs (*McAfee*, *TrustGo* and *Norton*) fall in this category (column "length" in Table I). We were not able to verify improper credential storage for these apps though. We tried the "recover lost password" but this only provided a reset link, not a clear-text password. Nevertheless there is no valid reason for MAVs to limit the length of passwords for more paranoid users.

Online rate limiting is a natural defensive measure against online guessing attacks. At the time of our study, we found that three products implemented it (*McAfee*, *Norton* and *Lookout*, column "online rate limiting" in Table I). For *Norton*, the lockout period did not work when we tested. Since our study finished, more MAV solutions have implemented rate-limiting in their web interface, but many still fail to enforce it in within the app. It is important to realise that while account locking might thwart an all-out targeted online guessing, a slower, distributed, throttled attack might still succeed [4]. Rate limiting and account locking also interact poorly with targeted smartphone theft: if prior to stealing a device, an attacker can lock her victim's account (or render its access slower), she can prevent him from remotely locking or wiping the stolen device.

Response from MAVs: MAVs that responded acknowledged these findings. They generally pointed out that usable authentication is challenging. Therefore we think this is an area worth investigation for future research.

V. App Configuration & User Interface

Without admin privileges, MAVs cannot take advantage of built-in lock and wipe features; yet admin privileges must be granted explicitly by users. It is commonly accepted that it is hard for users to configure and use security software safely [5], [6], [7], [8], and this can be even harder for small-screen devices. Therefore it is important for MAVs to warn, guide and inform users accordingly.

Findings: As shown in Table I, only four MAVs warn users if they do not run as admin (column "in-app warning"). For *Avira*, if a user clicks the warning but does not subsequently grant admin privileges, the warning disappears for ever. *Norton*, *Avast* and *TrendMicro* go further in the wrong direction: they display "Anti-Theft is on", "You are protected" and "Device now protected" respectively, either in the app or the online interface, even when apps do not run as admin. The column "in-app flow" (Table I) refers to a MAV that automatically launches the Android Settings view for granting admin privileges, when a user visits the relevant "Anti-Theft feature" menu of the app. This reduces the chance of misconfiguration, since a user need not struggle with finding the relevant Settings option. Three apps do take this approach. However, we found that if a user has granted admin privileges, but later decides to remove them, apps generally fail to notify "out-of-band" (i.e. outside the app – this is important as users do not interact with AV apps regularly and would miss in-app notifications).

Similarly, none of the MAVs warn users about app misconfiguration in the web interface either (column "web warning"). Furthermore, most of them *misinform users about the results of a remote wipe and lock*: they merely claim it is "successful" even if the phone app does not run as admin. We will see in Section VI and Section VII that the lack of admin privileges significantly reduces the reliability of anti-theft functions. A few solutions do give information

[5]code.google.com/p/android-apktool/

TABLE I
MAV IMPLEMENTATIONS: GENERAL FINDINGS

	User interface			Account Password				SSL	
	web warning	in-app warning	in-app flow	online rate limiting	length	special characters	only strong	certificate validation	pinning
AVG	✗	✗	✗	✗	$6 \leq l < \infty$	✓	✗	✓[2]	✗
Lookout	✗	✗	✗	$1hour$ wait if 13 incorrect attempts[0]	$5 \leq l < \infty$	✓	✗	✓[2]	✗
Avast	✗	“Issues”[1]	✗	✗	$7 \leq l < \infty$	✓[0]	✗	✓[2]	✗
Dr.web	n/a	✗	✓	n/a	$4 \leq l < \infty$	n/a	n/a	n/a	n/a
Norton	✗	✗	✗	CAPTCHA and account locked[0]	$6 \leq l \leq 50$	✓	✗	✓[2]	✗
McAfee	✓[1]	orange flag	✓	$1hour$ reset password after 5 attempts	$8 \leq l \leq 32$	✗	✗	✓[3]	✗
Kaspersky	✗	“issues”	✓	✗	$8 \leq l < \infty$	✓	✓	✓[2]	✓
TrustGo	✗	✗	✗	✗	$6 \leq l \leq 15$	✗	✗	✓	✓
TrendMicro	✗	✗	✗	✗	$8 \leq l < \infty$	✗	✗	✗[2]	✗
Avira	✗	“Action Required!”	✗	✗	$5 \leq l < \infty$	✗	✗	✓	✗

✗means absence of protection, whereas ✓is the opposite. [0]implementation problems are reported in the text of the paper.
[1]warning provided but in a specific tab. [2]use of http at registration time or startup.
[3] certificate validation during registration is broken.

in the web interface but only after a wipe command is issued, which is generally too late because this happens when the device is already lost.

Given the lack of guidance and warning provided by MAVs to users, it is likely many users will run an insecure setup. Throughout the following sections, we therefore take account of this observation and highlight several consequences.

Response from MAVs: MAVs that responded acknowledged these findings. They all say they are taking actions to improve the UI of their app.

VI. LOCK IMPLEMENTATIONS AND EFFECTIVENESS

One anti-theft option for users to protect their data on stolen devices is to remotely lock the screen, generally through a web page provided by MAVs. In the following sections, we assume that the Android Debug Bridge (adb) is disabled or protected in the device Settings (if not, then a thief can get an interactive shell and access user data as described in Section II-C). As users may have improperly configured their MAV (Section V), we study outcomes whether it runs as admin or not. The following sections present the many attack vectors we discovered during our study.

A. Removal of MAVs & API Misuse

Finding 1: When a MAV does not run as admin, it must resort to ad-hoc solutions to implement the screen lock, and must additionally ensure the app launched at device boot time. At boot time, there is a race condition, in that the custom lock screen should appear fast enough to prevent a thief from un-installing the MAV. We found that for four MAVs, the custom screen lock does not show up quickly enough and can therefore be un-installed (column “race protection” in Table II). By rebooting a stolen device and winning the race, a thief can remove the app and prevents the owner from remotely protecting the device.

Finding 2: By design (Section II-C), the Safe mode lets a thief un-install a MAV so long as the app does not have admin privileges. A race condition is no longer needed in this case, and all MAVs incorrectly configured by users (i.e. non-admin) are therefore vulnerable. Again, a user who remotely locks his screen is generally left clueless about the problem since MAV web interfaces do not provide details (Section V).

Finding 3: An admin MAV should invoke the built-in screen lock, which eliminates race conditions and safe mode bypass on the condition that it requests the `force-lock` policy and calls the relevant APIs (e.g. *lockNow()*). We found that four MAVs *misuse the security API*, in that they do not enable the default screen lock even though they request the `force-lock` policy (*Dr.web*, *McAfee*, *TrustGo* and *TrendMicro*). They can therefore be bypassed via Safe mode (Table II, column “admin lock protection”). Although *Avast* properly enables the built-in lock, it can also be bypassed – we defer this discussion to Section VI-C.

Finding 4: Even if an admin MAV properly enables the built-in lock screen, a thief could un-install the app before a user has remotely locked his device – on the condition that he first manages to remove the admin privileges. We found that seven MAVs leave the removal of admin privileges unprotected (Table II, column “un-install protection”). Only three apps take account of this attack vector. *Kaspersky* enables the built-in screen lock on first run of the app; the device remains locked at all time without the need for remote activation, and so is immune to this attack. *McAfee* and *Avast* prompt a thief with a PIN if he attempts to remove admin privileges. *Avast*’s lock (including the anti-removal lock) can be bypassed, but we defer the discussion to Section VI-C.

McAfee misuses the Android API so its anti-removal lock can be circumvented by re-booting into Safe mode, and the app subsequently removed. *McAfee*’s code for handling the removal of admin privileges is illustrated in Fig. 3. It uses the callback function *onDisabled()* to be notified

TABLE II
MAV REMOTE WIPE AND LOCK FUNCTIONS

	Wipe			Lock						
	data partition	primary SD	secondary SD	lock	on-device rate limiting	un-install protection	admin lock protection	non-admin lock protection	counter reset protection	race protection
AVG	admin	admin,format, unlink	unlink[1]	user-selected 4-6-digit PIN or “alpha” password	✗	✗	bp:brute-force	bp:Safe mode	n/a	✓
Lookout	admin	file overwrite, unlink	file overwrite, unlink	random 4-digit PIN	$T = \frac{N}{3} \times 1min^0$	✗	✓[0]	bp:Safe mode	bp:remove battery	✗
Avast	admin	unlink,format, admin	unlink,format	user-selected 4-6-digit PIN	✗	bp:GSM BTS	bp:GSM BTS	bp:Safe mode	n/a	✗
Dr.web	admin	admin,unlink	unlink	4-char password	✗	✗	bp:Safe mode	bp:Safe mode	n/a	✓
Norton	admin	admin	format or unlink[2]	random 4-digit PIN	optional wipe after 10 attempts	✗	✓	bp:Safe mode	✓	✓
McAfee	admin	admin,unlink, format[3]	unlink,format[3]	user-selected 6-digit PIN	$1hour$ wait after 10 attempts	bp:Safe mode	bp:Safe mode	bp:Safe mode	bp:remove battery	✗
Kaspersky	admin	unlink	unlink	user-selected 4-16-digit PIN	✗	n/a	built-in lock	✓	n/a	✓
TrustGo	admin	unlink	unlink	web password	$30min$ wait after 5 attempts	✗	bp:Safe mode	bp:Safe mode	✓	✓
TrendMicro	admin	format,unlink	format,unlink	web password	✗	✗	bp:Safe mode	bp:Safe mode	n/a	✗
Avira	admin	unlink	unlink,format	user-selected 4-digit PIN	unlock with web only after 3 attempts	✗	✓	bp:Safe mode	✓	✓

✗means absence of protection, whereas ✓is the opposite. “bp:method” means that the protection is present but can either be bypassed using “method” or its effectiveness reduced using “method”. Note that vendors’ customisations may allow bypass of properly implemented protections. [0]implementation problems are reported in the text of the paper.
[1]only if the app is not an admin and formatting of the internal (primary) SD card fails. [2]only if the app is not an admin. Format if $SDK >= 17$, *unlink* files otherwise. [3]formats if a tablet.

of admin changes: when this function is called by the Android framework, *McAfee* locks the screen. Unfortunately, on Gingerbread devices (v2.3.x), the *onDisabled()* function is called only *after* the admin privileges are disabled. As a result, the subsequent call to lock the screen cannot make use of the built-in lock screen and must be implemented by the app itself. Therefore when rebooting into Safe mode, the lock screen does not show up since apps are disabled in this mode (Section II-C). At this point, the app can also be removed as it no longer has admin privileges.

```
public class McAfeeReceiver extends DeviceAdminReceiver {

 public void onDisabled(Context paramContext,
                        Intent paramIntent){
  [...] // removed
  displayLockScreen();
 }
}
```

Fig. 3. Code of McAfee’s anti-removal screen.

Further investigation reveals that the *onDisabled()* function is called *before* admin privileges are dropped on subsequent Android versions we tested (ICS (v4.0.x), Jelly Bean (v4.[1-3]). This problem is aggravated because *the Android documentation is incorrect*. More specifically, it is oblivious to differences between versions[6]. It only states that the *onDisabled()* function is “called prior to the administrator being disabled” without specifying the relevant Android versions. We used the Internet Archive[7] to trace changes to the API description. We found the API documentation was also incorrect at the time when Gingerbread was the most recent Android version (Dec 2010 – Oct 2011). Another function, *onDisabledRequested()*, is called *before* admin privileges are dropped for all Android versions we tested. Therefore it is more reliable, but MAV apps currently do not use it.

There are also usability problems associated with the protection of app removal. As users tend to forget passwords/PINs, a user who genuinely attempts to un-install a MAV may be presented with a (PIN) screen lock he cannot remove. Therefore MAVs must provide additional information in their lock screen, e.g. to guide users how to reset the PIN. But the *default Android lock screen has certain limitations*, in that it cannot be customised. As a result, additional information cannot be provided with the default lock screen, and MAVs resort to implementing their own lock screen; which we know can be bypassed through race conditions or Safe mode. Android Lollipop (v5.0) provides a new function (*startLockTask()*[8]) for “screen pinning”, that is, to “temporarily restrict users from leaving your task”. However this would still be by-passable in Safe mode. Therefore we think Android would benefit from a customisable lock screen.

Response from MAVs: Problems were generally acknowledged. One company argued that the Safe mode bypass of their lock screen was “low risk” because “third-party apps do not run in Safe mode”. This is incorrect. First, a thief read user’s emails manually and gain access to credentials for other websites – e.g. contained in emails or through a reset link sent to their Inbox. Second (Section II-C), even though disabled in the user interface, apps can be installed and launched via a shell in Safe mode to automate the process.

One company argued that they do not enable the built-

[6]developer.android.com/reference/android/app/admin/DeviceAdminReceiver.html
[7]web.archive.org/web/20100501000000*/https://developer.android.com/reference/android/app/admin/DeviceAdminReceiver.html
[8]https://developer.android.com/about/versions/android-5.0.html

in lock because it may violate the Google policy. This states that "an app downloaded from Google Play [...] must not make changes to the user's device outside of the app without the users knowledge and consent". We said we disagree because when remotely locking his device, a user is implicitly giving his consent.

Regarding the lack of un-installation protection, we have received feedback from few developers. Many apps concerned with this problem did not have a proper contact/email for bug reports. We had to use the "customer support" portal instead. Customer support assured us they would forward our report to the engineers but we have not heard from them in most cases.

B. Rate Limiting

Finding 1: Admin MAVs that properly enable the built-in screen lock generally also overlay their own lock on top of the default lock screen. This is used to customise the look-and-feel or offer additional information. For example, MAVs sometimes want to provide an email address to contact the phone owner if the device is lost. This overlay is also a source of problems. The built-in screen lock, on most devices, enforces a $30sec$ wait period after 5 failed PIN attempts. Unfortunately half the MAVs do not have rate limiting protection on their custom screen-lock (column "on-device rate limiting"), thereby annihilating the protection of the built-in one. This makes brute force practical since MAVs accept common weak PINs like "1111" or "1234". Only *Kaspersky* warned us to use at least 7 digits, yet it still accepted weak PINs. This issue again highlights *a limitation of the default lock screen*, in that it cannot be customised enough, therefore MAVs have tried to find ways around this. This suggests that the default lock screen would benefit from an additional admin "lock rate limiting" policy for apps to express requirements such as "enforce an X sec wait after Y successive incorrect PINs".

Finding 2: Some MAVs apps do enforce stronger rate-limiting policies than the built-in default (column "on-device rate limiting"). For *Lookout* however, we found that it could be circumvented on our Galaxy S Plus even when the app runs as admin: by clicking the Home button, a thief can navigate away from the *Lookout*'s custom screen lock, and benefit from the lower rate limiting of the built-in one. In Safe mode, a non-admin MAV's custom screen lock does not show up, so if the built-in lock has less protection, a thief can use that instead – on certain devices such as the Samsung Galaxy S Plus, the built-in screen has no rate limiting.

Finding 3: In order to enforce rate-limiting after too many incorrect PIN attempts, MAVs must save a retry counter. For *Lookout*, we found that removing the battery resets the retry counter, and for *McAfee* it decreases it by one. This attack is not always practical, as a thief must reboot the phone but the boot time may exceed the wait period enforced by a MAV. In *Lookout*'s case, it significantly decreases the wait period – which increases linearly with the number of incorrect PIN attempts (Table II, column "on-device rate-limiting").

Finding 4: If an attacker knows the username of his victim, he could also leverage the lack of rate limiting in the online interface to unlock the screen remotely (Section IV). This is mostly true in targeted attacks. In non-targeted ones, it may still be the case because some MAVs, by default, display the owner's email address in their custom lock (e.g. *Avira* and *TrendMicro*). This unintentionally leaks the username necessary to perform an online dictionary attack.

MAV Response: Our findings were acknowledged. One company responded to the "Finding 4" by masking some of the characters in the email address.

C. Network-level Attacks

Finding 1: *Avast*, downloaded more than 100M times on Google Play, has an option (including in its "anti-removal" lock) that lets thieves send temporary unlock PINs via SMS to a (compulsory) contact pre-configured by the genuine device owner. This function is accessible in the custom screen lock. When used, the stolen device sends a temporary PIN – in clear – to a trusted contact. It is known that GSM only authenticates a phone to the network, but not the network to the phone. Therefore it is possible to have the phone connect to a rogue GSM station – a.k.a. Base Transceiver Station (BTS) – to recover traffic. Because phones usually fall back to GSM in the absence of 3G, a thief could set up a rogue BTS to intercept the temporary PIN and bypass the lock screen to remove the app.

Similarly, *Dr.web* locks/unlocks devices through commands sent via SMS from a trusted contact. If the SMS sender is a pre-configured trusted "buddies", a password is not required (this is enabled by default). In a targeted scenario where a trusted buddy's phone number is known, a thief could spoof the SMS's sender ID to bypass the password. One could use online services, a rogue BTS station or a femtocell.

Finding 2: Recall from Section II-D that MAVs generally keep a TCP connection to a server in order to receive remote commands to be executed on the device. An attacker who can impersonate as a MAV server could therefore send an unlock command to the phone app. MAVs generally use TLS to secure their connection and properly verify their server's certificate but do not implement pinning (Table I, column "SSL"). *TrendMicro* however did not properly validate the CN of the certificate. It accepted any domain (i.e. CN) so long as it was signed by a trusted CA. *McAfee* improperly authenticated its server during account registration: the app accepted selfsigned certificates as well as certificates signed by a trusted CA but for a different domain (i.e. CN). Even though they used TLS, *Avast*, *AVG*, *Lookout*, *Norton*, *Kaspersky* and *TrendMicro* additionally made some non-encrypted requests. Instead of maintaining a TCP connection to their server, certain MAVs such as *Avira* re-use the Google connection and push commands to the phone through Google push APIs. We found that the Google service did not implement certificate pinning. On a side note, *TrustGo*'s user web interface used a combination of http and https, making cookie recovery trivial for a network attacker.

MAV Response: The latest version of their apps fixes certificate validation issues – not necessarily pinning. We have not received feedback regarding unprotected PINs and commands sent via carriers' networks (Finding 1).

D. Vendor and Customised Android Failures

Even if anti-theft functions are properly implemented by MAVs, customisations by vendors may allow an attacker to bypass their protection.

Finding 1: We refer to "charging mode" as the mode in which a device is, when it is switched off and plugged into a power supply. We found that certain phones boot up a headless Android and expose a debugging interface when booted in charging mode. For example, the LG L7 running Jelly Bean (v4.1.2) exposes adb: the data partition is mounted and one folder is writeable, so an attacker could push code and exploits. The Samsung Galaxy S Plus also exposes adb but we did not have enough permissions to push files; yet it remains a concern.

Finding 2: On phones with open Bootloaders, a thief can install custom upgrade: this bypasses admin MAV screen locks and the built-in one. Thieves could therefore install forensics software to recover personal data. Protected Bootloaders may also contain logic errors or vulnerabilities[9]. We found that on the Google Nexus running Android v4.2, unlocking the Bootloader did not erase the data partition; on the HTC Desire C, a Bootloader unlock wiped only the data partition; and on the HTC Desire S, there was a Bootloader option to read arbitrary files. When booted in Recovery mode, the Samsung Galaxy S2 also exposes adb, and the LG L7 further has a writeable folder. The latter even warns "Using this mode, service center can back up and recover your data".

Finding 3: Users who want more control over their device may install a custom Recovery from which they gain additional functionalities, such as root access, the ability to install custom Android builds, custom UI themes, etc. As the Recovery has unrestricted access to Android binaries and data partitions (Section II-C), it is important to prevent unauthorised users (i.e. thieves) from accessing it. We found this is not always the case. For example, prevalent Recoveries such as CWM or TWRP[10] often expose unprotected options that allow flashing arbitrary software via USB even when the screen is locked in the Android OS. In our tests, we could push arbitrary Android updates via the following options. The "zip updates" option accepts arbitrary updates from files present on the primary SD card, so an attack is feasible on phones with a removable primary SD card. The "adb sideload" accepts updates sent in-band via USB. Newer Android versions protect adb-via-USB using a public-key authentication scheme tied to the owner's computer (though bypassable on certain devices[11]). This security feature was not enforced in the custom recoveries we tested: adb generally remained available and unprotected even when not enabled in the Android Settings, and the signature verification on updates could be toggled off manually.

Finding 4: Vendors also have desktop software that one can use to backup or transfer files from a smartphone via USB (like Samsung Kies or HTC Sync Manager). We found that the activation of the HTC Sync Manager automatically enables the Android Debug Bridge (adb) on the device – probably because it piggy-backs on its protocol. Some HTC devices keep adb on even when the screen is locked, allowing a thief to get an interactive shell on the device via USB. We leave a comprehensive security analysis of vendors' software for future research.

E. Misc

Android has in the past been victim of PIN bypass vulnerabilities that let a thief access partially, or fully, a device's user interface and data. It may be because of third-party apps[12] or vendor customisations[13]. These were out of scope of this study.

Even when a lock cannot be bypassed, current Android phones allow a thief to Factory Reset the device from the Recovery or Bootloader mode. This deletes the data, but not always securely (Section II-B). Therefore a thief could alternatively Factory Reset a device and use forensic tools to recover insecurely-deleted data on devices with a flawed Factory Reset function. This may change in newest Android devices with the introduction of a "kill-switch" function introduced to curb phone theft. This was out of scope of our study.

Some HTC devices have a DIAGnostic that can be accessed via specially-formatted SIMs or SD cards known as "gold cards". It gives access to vendors' "repair menus". On our devices, data was wiped, but this may not be the case on other models.

We discuss hardware attacks and baseband attacks in Section VIII.

F. Encryption to the Rescue

The use of Full Disk Encryption (FDE) has the potential to mitigate many of the attacks when the phone is powered down, so long as the encryption key cannot be recovered. On Android devices that support FDE, the encryption key is stored encrypted with a key derived from a salt and a user-provided PIN (or password). This encrypted blob is referred to as the "crypto footer" in the AOSP source code. We found that if the PIN entered by a user is invalid, Android reads a

[9] www.codeaurora.org/projects/security-advisories/fastboot-boot-command-bypasses-signature-verification-cve-2014-4325 and www.codeaurora.org/projects/security-advisories/incomplete-signature-parsing-during-boot-image-authentication-leads

[10] www.clockworkmod.com – www.teamw.in/project/twrp2

[11] labs.mwrinfosecurity.com/advisories/2014/07/03/android-4-4-2-secure-usb-debugging-bypass and github.com/secmobi/BackupDroid

[12] seclists.org/fulldisclosure/2013/Jul/6 and www.bkav.com/top-news/-/view_content/content/46264/critical-flaw-in-viber-allows-full-access-to-android-smartphones-bypassing-lock-screen

[13] shkspr.mobi/blog/2013/03/new-bypass-samsung-lockscreen-total-control

TABLE III
WIPE IMPLEMENTATIONS WITHOUT ADMIN PRIVILEGES

	result timing	disable data sync	media	SMS/MMS	contacts	accounts	accounts password	browser history	browser credentials	call logs	WiFi	cache	VPN	own data
AVG	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✗	✗	✗	✗
Lookout	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✗	✓	✗	✗
Avast	✓	✗	✓	✓	✓	✗	✗	✓	✗	✓	✗	✗	✗	✗
Dr.web	n/a	✗	✗*	✓	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗
Norton	✓	✗	✗*	✓	✓	✗	✗	✓	✗	✓	✗	✗	✗	✗
McAfee	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗
Kaspersky	✗	✓	✗*	✓	✓	✗	✓	✗	✗	✓	✗	✗	✗	✗
TrustGo	✗	✗	✗*	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗
TrendMicro	✗	✗	✓	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✓
Avira	✓	✗	✗*	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

*data is nevertheless removed by deleting the corresponding files on the SD card

retry counter from the crypto footer, increments it and writes it back. By monitoring power consumption, an attacker could detect the write back, and cut the power supply to circumvent the write[14]. This attack was well known in the smartcard community in the 1990s; by fifteen years ago, it was standard practice to increment the retry counter before each PIN prompt and reset it only if a correct PIN is entered [9].

Another problem stems from the "Fast boot" option on HTC devices. It is enabled by default, and when powered down, a device transitions into a hibernation/sleeping state so as to speed up boot time. When powered up, the PIN is not prompted so it may be stored in RAM or on disk. Failure to disable the fast boot option may undermine the benefit of FDE. We leave this for future research.

VII. WIPE IMPLEMENTATIONS

A. General Results

The second option provided by anti-theft functions is the "remote wipe". We study MAV wipe implementations with admin privileges granted or not.

Admin MAVs: Table II (column "data partition") shows that all MAVs use the admin API to sanitise the data partition. For external storage though, implementations differ.

Half of them use the admin API to sanitise the primary SD card (the internal SD card on our Galaxy S Plus). If the sanitisation fails, they implement "backup" mechanisms. For instance *AVG*, *Avast* and *McAfee* format the partition and *unlink* all files; this does not provide logical sanitisation.

The other half of MAVs *misuses the Android security API* in the sense that they do not pass the flag *WIPE_EXTERNAL_STORAGE* to sanitise external storage, and this even when apps have been granted admin privileges. Instead they resort to ad-hoc mechanisms: they generally format the partition and/or *unlink* files; this does not provide logical sanitisation. We think the misuse of the Android API may be caused by *misleading/outdated Android documentation*: it states that the flag parameter to the *wipeData(int flag)* function "must be 0"[15].

[14]Limitations: (*i*) a full kernel is running with certain userspace processes, (*ii*) the OS may cache (and therefore delay), the write-back to non-volatile storage, (*iii*) rebooting the device may be too slow.

[15]https://developer.android.com/guide/topics/admin/device-admin.html

The Android API currently *does not expose an API to securely sanitise the secondary SD card*. Therefore, all MAVs implement their own mechanisms to sanitise it, as a result of which data is always recoverable. We note that even when used properly, the admin API does not ensure that logical sanitisation is available on the device (Section II-B).

Non-Admin MAVs: When MAVs do not run as admin, they must resort to other mechanisms to sanitise partitions, e.g. by using Android APIs (Section II-D). Recall that this situation is likely to arise because of UI issues highlighted in Section V. In this case, MAVs attempt to delete data as shown in Table III; but this does not provide logical sanitisation. Furthermore, we found that none of the MAVs manage to remove the primary account using Android APIs. Some of them try to, but fail because the OS does not seem to allow it: this failure is never reported to a user in the online interface. Android *does not appear to expose an API to delete browser, WiFi and VPN credentials*, and no MAVs delete them.

A thief could also attempt to recover MAV credentials on the phone in order to access online backups made by a user. We found that only *TrendMicro* attempts to delete its "own data" (Table III).

We also found that three MAVs display success for a remote wipe in their web interface, even if the wipe has yet to be performed (Table III, column "result timing"). This is because on reception of the wipe command from their server, MAV apps acknowledge it: some web interfaces interpret this as wipe success. All apps, to some extent, *misinform users about wipe results*, merely claiming that a remote wipe succeeds when errors occur, like when an exception is thrown, when formatting fails, or when a file cannot be *unlink*ed. They all fail to give accurate details about the wipe results in general. *McAfee*, *Avast* and *Avira* fare slightly better because they indicate that the wipe cannot use the built-in Factory Reset if admin privileges are not granted, but this is too late when the device is already lost.

B. Case Studies of Lookout and Avast

Lookout and *Avast* both have unique implementations for sanitising external storage. *Lookout* sanitises external storage by overwriting files before *unlink*ing them. Its developers implicitly assume that the file system overwrites files "in-place". We tested this hypothesis by creating 1000 files on the primary SD card of our Galaxy S Plus (FAT-formatted). We overwrote the files, and found that more than 90% of them were not updated "in-place" by the file system – and thus were recoverable. We ran the same experiment with *Lookout*'s remote wipe, and obtained similar results.

Avast has a special option called "thorough wipe" to sanitise external storage. This creates a 1MB file and overwrites it 1000 times with $0s$ (all values are hardcoded in the app). Its developers implicitly assume that the file system does not update files "in-place", which would ensure that 1GB of space is overwritten. We showed this to be mostly true on our Galaxy S Plus's primary SD card formatted with FAT. However, on an ext4-formatted partition, we found that more than 99% of files were updated "in-place" on an HTC One S. Hence, mostly the same blocks, spanning 1MB, are overwritten. Furthermore, all devices we have encountered, have at least 2GB of primary SD card, while some have up to 25GB (HTC One X). As a result, even if 1GB of data was indeed overwritten, it would not represent the entire partition. Note that the ext4 filesystem may also use compression which will further reduce the amount of data overwritten.

These two examples highlight how *the security permission model* can sometimes backfire and have a negative impact on security: because third-party apps cannot legitimately gain high privileges (i.e. root), they cannot overwrite an entire partition bit-by-bit, and therefore resort to unreliable methods to mitigate a flawed Factory Reset (Section II-B).

MAV Response: We have received feedback only from one of the two companies. The dev team said they are trying to improve the reliability of remote wipe without impacting usability.

C. Inherent Problem of Remote Wipe

We found an inherent timing problem between the time of wipe and the time of success displayed to users in the web interface: we call this problem the "Time-Of-Wipe-Time-Of-Success (TOWTOS)" attack. Concretely, this occurs when a wipe completion is reported to the user in the web interface before the wipe effectively takes place on the phone, and the wipe subsequently fails. The failure could be caused by the removal of the battery by a thief, by the phone being shutdown, or by a TLS truncation attack [10] by a thief attempting to desynchronise the phone app and the server. This is currently inevitable for an MAV that performs an admin wipe because it must report to the user before it is itself wiped. *Lookout* aggravates this problem by launching a $10sec$ timer before launching an admin wipe. The TOWTOS highlights *another limitation of the current Android admin API*, in that it is not possible for an app to report results to its web service after a wipe completes.

VIII. Discussion

We have highlighted three main failures in remote data protection functions: (*i*) the misuse of Android security APIs by MAVs, (*ii*) the limitation of Android APIs and permission model, (*iii*) vendor customisation issues. One attack we discarded till now is the use of Faraday bags. There are two lines of thought to mitigate it. The first involves an online server that is contacted during user authentication to a device, the approach is taken by CleanOS [11], KeyPad [12] and other theft-resilient solutions [13]. However these can severely impact usability by (*i*) degrading responsiveness of the device during authentication when there is poor network coverage, and (*ii*) by locking a genuine user out if a server is DoSed, unavailable or if there is no network coverage.

Therefore solutions that solely rely on the user and device seem the only viable options. Authentication based on user behaviour [14], swipe characteristics [15], context [16], typing characteristics [17] and app usage [18] are appealing but usually lack accuracy and speed as well as increasing power consumption. Solutions based on an extra device users carry [19] often impose extra cognitive load and are seen as a burden. Biometrics such as Apple's fingerprint are relatively fast and effortless and therefore appealing to average users on high-end devices.

Overall, we think that device-based solutions are more likely to offer usable protections against the attacks described in this paper. More importantly, given the limitations imposed by the Android API and the permission model, we think the only viable solutions are those driven by vendors themselves. They, only, can integrate their solutions seamlessly whilst taking full advantage of the platform and hardware features. The rise of wearable computing may also enable better authentication mechanisms if battery life improves; this is another avenue for future research to consider.

IX. Related Work

Cannon [20], Osborn [21], and Ossmann and Osborn [22] attempted to access personal data on smartphones with physical access to them, but did not study MAVs. Baseband attacks have been studied by Weinmann [23], Mulliner [24] and Solnik and Blanchou [25]. In particular, Solnik and Blanchou [25] demonstrated a complete bypass of the main OS lock screen on Android and iOS devices. They exploited Over-The-Air (OTA) update mechanisms used by carriers. Pereira and al. [26] discussed the security implications of non-standard AT commands exposed by certain Samsung phones via USB; these could be used to access personal data even on screen-locked devices. Mahajan et al. [27] used commercial software on 5 different devices to recover Viber and Whatsapp chats from unlocked Android smartphones. The forensics software requires adb to do storage acquisition. Müller *et al.* used a cold boot attack on Samsung Nexus devices to recover FDE keys from RAM. Their hack however requires an open (or unlocked) Bootloader to flash their custom forensics Recovery.

Hardware attacks are usually more expensive, do not scale or require per-device knowledge and skills, yet they are feasible. Unprotected JTAG[16] and UART [22] access are recent examples targeted at smartphones. Attaching a bus monitor to monitor data transfers between a CPU and main memory is also feasible; commercial solutions are available [28], [29]. DMA attacks have been showcased by Becher *et al.* [30], Boileau [31], Witherden [32] and Piegdon [33], although not on smartphones. To increase the cost of these attacks, Copl *et al.* [34] use ARM-specific mechanisms to keep application code and data on the System-on-Chip (SoC) rather than in DRAM.

To the best of our knowledge, this paper is the first comprehensive study of consumer grade anti-theft functions

[16]www.forensicswiki.org/wiki/JTAG_Samsung_Galaxy_S3_%28SGH-I747M%29 – copgeek018.wordpress.com/2012/04/03/157/

provided by mobile anti-virus apps on the Android platform.

X. Conclusion and Future Work

We studied consumer grade protections against unauthorised access to personal data on stolen Android smartphones. We further highlighted the market and technical fragmentation which means that there is no consistent security guarantee across devices.

We unveiled critical failures on MAV remote lock and remote wipe functions. In addition to the limitations imposed by the Android security model and APIs, these are caused by questionable MAV designs and vendor customisations.

Future research could investigate in more detail the level of security provided by smartphones when capable attackers have physical access to them. This could encompass new attack vectors introduced by HTC's "Fast boot" option, by device-management vendor software, and by the new "kill-switch" to be introduced in the USA to curb smartphone theft.

XI. Acknowledgements

The authors thank the anonymous reviewers for their valuable suggestions and comments. This work was supported by the Samsung Electronics Research Institute (SERI) [grant number RG67002].

References

[1] L. Simon and R. Anderson, "Security analysis of android factory resets," in *4th Mobile Security Technologies Workshop (MoST)*, 2015.

[2] M. Y. C. Wei, L. M. Grupp, F. E. Spada, and S. Swanson, "Reliably erasing data from flash-based solid state drives.," in *FAST*, vol. 11, pp. 8–8, 2011.

[3] J. Bonneau and S. Preibusch, "The password thicket: Technical and market failures in human authentication on the web.," in *WEIS*, 2010.

[4] M. Javed and V. Paxson, "Detecting stealthy, distributed ssh brute-forcing," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 85–96, ACM, 2013.

[5] S. Clark, T. Goodspeed, P. Metzger, Z. Wasserman, K. Xu, and M. Blaze, "Why (special agent) johnny (still) can't encrypt: A security analysis of the apco project 25 two-way radio system.," in *USENIX Security Symposium*, Citeseer, 2011.

[6] S. Fahl, M. Harbach, T. Muders, M. Smith, and U. Sander, "Helping johnny 2.0 to encrypt his facebook conversations," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, (New York, NY, USA), pp. 11:1–11:17, ACM, 2012.

[7] A. Whitten and J. D. Tygar, "Why johnny can't encrypt: A usability evaluation of pgp 5.0," in *In Proceedings of the 8th USENIX Security Symposium*, 1999.

[8] S. Ruoti, N. Kim, B. Burgon, T. van der Horst, and K. Seamons, "Confused johnny: When automatic encryption leads to confusion and mistakes," in *Proceedings of the Ninth Symposium on Usable Privacy and Security*, SOUPS '13, (New York, NY, USA), pp. 5:1–5:12, ACM, 2013.

[9] R. Anderson and M. Kuhn, "Tamper resistance: A cautionary note," in *Proceedings of the 2Nd Conference on Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2*, WOEC'96, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 1996.

[10] B. Smyth and A. Pironti, "Truncating tls connections to violate beliefs in web applications," in *USENIX Workshop on Offensive Technologies (WOOT 13)*, 2013.

[11] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda, "Cleanos: Limiting mobile data exposure with idle eviction," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 77–91, USENIX Association, 2012.

[12] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy, "Keypad: An auditing file system for theft-prone devices," in *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, (New York, NY, USA), pp. 1–16, ACM, 2011.

[13] P. D. MacKenzie and M. K. Reiter, "Networked cryptographic devices resilient to capture," in *2001 IEEE Symposium on Security and Privacy, Oakland, California, USA May 14-16, 2001*, pp. 12–25, 2001.

[14] E. Shi, Y. Niu, M. Jakobsson, and R. Chow, "Implicit authentication through learning user behavior," in *Proceedings of the 13th International Conference on Information Security*, ISC'10, (Berlin, Heidelberg), pp. 99–113, Springer-Verlag, 2011.

[15] S. M. Kolly, R. Wattenhofer, and S. Welten, "A personal touch: Recognizing users based on touch screen behavior," in *Proceedings of the Third International Workshop on Sensing Applications on Mobile Phones*, PhoneSense '12, (New York, NY, USA), pp. 1:1–1:5, ACM, 2012.

[16] E. Hayashi, S. Das, S. Amini, J. Hong, and I. Oakley, "Casa: Context-aware scalable authentication," in *Proceedings of the Ninth Symposium on Usable Privacy and Security*, SOUPS '13, (New York, NY, USA), pp. 3:1–3:10, ACM, 2013.

[17] E. Maiorana, P. Campisi, N. González-Carballo, and A. Neri, "Keystroke dynamics authentication for mobile phones," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, (New York, NY, USA), pp. 21–26, ACM, 2011.

[18] H. Khan and U. Hengartner, "Towards application-centric implicit authentication on smartphones," in *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, HotMobile '14, (New York, NY, USA), pp. 10:1–10:6, ACM, 2014.

[19] F. Stajano, "Pico: No more passwords!," in *Proceedings of the 19th International Conference on Security Protocols*, SP'11, (Berlin, Heidelberg), pp. 49–81, Springer-Verlag, 2011.

[20] T. Cannon, "Into the droid: Gaining access to android user data." https://www.defcon.org/images/defcon-20/dc-20-presentations/Cannon/DEFCON-20-Cannon-Into-The-Droid.pdf.

[21] K. K. Osborn, "Physical drive-by downloads." http://www.irongeek.com/i.php?page=videos/derbycon2/1-2-9-kyle-kos-osborn-physical-drive-by-downloads.

[22] M. Ossmann and K. Osborn, "Multiplexed wired attack surfaces," in *BlackHat USA*, 2013.

[23] R.-P. Weinmann, "Baseband attacks: Remote exploitation of memory corruptions in cellular protocol stacks.," in *WOOT*, pp. 12–21, 2012.

[24] C. Mulliner, N. Golde, and J.-P. Seifert, "Sms of death: From analyzing to attacking mobile phones on a large scale.," in *USENIX Security Symposium*, 2011.

[25] M. Solnik and M. Blanchou, "Cellular exploitation on a global scale: The rise and fall of the control protocol," in *Black Hat USA*, 2014.

[26] A. Pereira, M. Correia, and P. Brandão, "Usb connection vulnerabilities on android smartphones: Default and vendors customizations," in *Communications and Multimedia Security*, pp. 19–32, Springer, 2014.

[27] A. Mahajan, M. Dahiya, and H. Sanghvi, "Forensic analysis of instant messenger applications on android devices," *arXiv preprint arXiv:1304.4915*, 2013.

[28] E. Solutions, "Analysis tools for ddr1, ddr2, ddr3, embedded ddr and fully buffered dimm modules." http://www.epnsolutions.net/ddr.html.

[29] F. System, "Ddr2 800 bus analysis probe." http://www.futureplus.com/download/datasheet/fs2334_ds.pdf.

[30] C. N. K. Michael Becher, Maximillian Dornseif, "Firewire - all your memory are belong to us," in *Proceedings of the CanSecWest Applied Security Conference*, 2005.

[31] A. Boileau, "Hit by a bus: Physical access attacks with firewire," in *Proceedings of the Ruxcon Conference*, 2006.

[32] F. Witherden, "Memory forensics over the ieee 1394 interface." https://freddie.witherden.org/pages/ieee-1394-forensics.pdf, 2010.

[33] D. R. Piegdon, "Hacking in physically addressable memory - a proof of concept," in *Seminar of Advanced Exploitation Techniques*, 2006.

[34] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman, "Protecting data on smartphones and tablets from memory attacks," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, (New York, NY, USA), pp. 177–189, ACM, 2015.