ATtention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem

Dave (Jing) Tian*[1], Grant Hernandez[1], Joseph I. Choi[1], Vanessa Frost[1], Christie Ruales[1], Patrick Traynor[1], Hayawardh Vijayakumar[2], Lee Harrison[2], Amir Rahmati[2,3], Michael Grace[2], and Kevin R. B. Butler[1]

[1]Florida Institute for Cybersecurity (FICS) Research, University of Florida, Gainesville, FL, USA
{*daveti,grant.hernandez,choijoseph007,vfrost,cruales,traynor,butler*}*@ufl.edu*

[2]Samsung Research America, Mountain View, CA, USA
{*h.vijayakuma,lee.harrison,amir.rahmati,m1.grace*}*@samsung.com*

[3]Department of Computer Science, Stony Brook University, Stony Brook, NY, USA

Abstract

AT commands, originally designed in the early 80s for controlling modems, are still in use in most modern smartphones to support telephony functions. The role of AT commands in these devices has vastly expanded through vendor-specific customizations, yet the extent of their functionality is unclear and poorly documented. In this paper, we systematically retrieve and extract 3,500 AT commands from over 2,000 Android smartphone firmware images across 11 vendors. We methodically test our corpus of AT commands against eight Android devices from four different vendors through their USB interface and characterize the powerful functionality exposed, including the ability to rewrite device firmware, bypass Android security mechanisms, exfiltrate sensitive device information, perform screen unlocks, and inject touch events solely through the use of AT commands. We demonstrate that the AT command interface contains an alarming amount of unconstrained functionality and represents a broad attack surface on Android devices.

1 Introduction

Since their introduction, smartphones have offered substantial functionality that goes well beyond the ability to make phone calls. Smartphones are equipped with a wide variety of sensors, have access to vast quantities of user information, and allow for capabilities as diverse as making payments, tracking fitness, and gauging barometric pressure. However, the ability to make calls over the cellular network is a fundamental characteristic of smartphones. One way this heritage in telephony manifests itself is through the support of AT commands, which are designed for controlling modem functions and date to the 1980s [24]. While some AT commands have been standardized by regulatory and industry bodies [35, 42], they have also been used by smartphone manufacturers and operating system designers to access and control device functionality in proprietary ways. For example, AT commands on Sony Ericsson smartphones can be used to access GPS accessories and the camera flash [18].

While previous research (e.g., [20, 46, 47]) has demonstrated that these commands can expose actions potentially causing security vulnerabilities, these analyses have been ad-hoc and narrowly focused on specific smartphone vendors. To date, there has been no systematic study of the types of AT commands available on modern smartphone platforms, nor the functionality they enable. In effect, *AT commands represent a source of largely undocumented and unconstrained functionality.*

In this paper, we comprehensively examine the AT command ecosystem. We assemble a corpus of 2,018 smartphone firmware images from 11 Android smartphone manufacturers. We extract 3,500 unique AT commands from these images and combine them with 222 commands we find through standards to create an annotated, cross-referenced database of 3,722 commands. To our knowledge, this represents the largest known repository of AT commands. We characterize the commands based on the evolution of the Android operating system and smartphone models and determine where AT commands are delivered and consumed within different smartphone environments. To determine their impact, we test the full corpus of 3,500 AT commands by issuing them through the USB charging interface common to all smartphones. We execute these commands across 8 smartphones from 4 different manufacturers. We characterize the functionality of these commands, confirming the operation of undocumented commands through disassembly of the firmware images.

Our analysis of discovered AT commands exposes powerful and broad capabilities, including the ability to bypass Android security mechanisms such as SEAndroid in order to retrieve and modify sensitive information. Moreover, we find that firmware images from newer

*Dave began this project during an internship at Samsung Research America.

smartphones reinstate AT command functionality previously removed due to security concerns, causing those vulnerabilities to re-emerge. In short, we find that AT commands accessed through the USB interface allow almost arbitrarily powerful functionality without any authentication required. As such they present a large attack surface for modern smartphones.

Our contributions can be summarized as follows:

- **Systematic Collection and Characterization of AT Commands.** We develop regular expressions and heuristics for determining the presence of AT commands in binary smartphone firmware images, extracting AT commands into an annotated database that tracks metadata and provenance for each command. Our database and code are publicly available at `http://atcommands.org`.

- **Comprehensive Runtime Vulnerability Analysis.** We systematically test 13 Android smartphones and 1 tablet for exposure to the USB modem interface and find that 5 devices expose the modem by default while 3 other devices will do so if rooted. Using this interface, we comprehensively test all 3,722 AT commands to determine the effect of both standard and vendor-specific commands on individual devices. We characterize notable classes of commands that can cause vulnerabilities such as firmware flashing, bypassing Android security mechanisms, and leaking sensitive device information. We find that new smartphone platforms reintroduce AT command-based vulnerabilities that were purportedly previously patched.

- **Development of Attack Scenarios and Mitigations.** We demonstrate that previously-disclosed attacks targeting the lock screen [49], which required malicious application access, can be performed through a USB cable without requiring any code on the target phone. We demonstrate that arbitrary touchscreen events can be injected over USB. We discover multiple buffer overflow vulnerabilities and commands to expose the contents of `/proc` and `/sys` filesystems, as well as path traversal vulnerabilities. We even discover a method to "brick" and "unbrick" certain phones. We also discuss how mechanisms such as "charger" mode and SELinux policies only partially mitigate the threat that broadly accessible AT commands can pose to smartphone platforms.

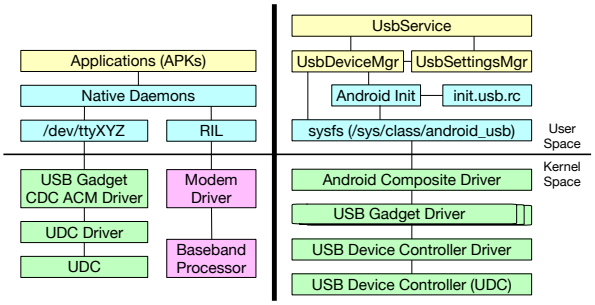


Figure 1: Android USB architecture diagrams. The left shows an Android device behaving like a USB modem when connected with a host machine and the right is an overview of the Android USB stack.

2 Background

2.1 AT Commands

First developed by Dennis Hayes in 1981, the AT (ATtention) command set comprises commands that predominantly start with the string "AT" [16]. The AT command set rapidly became an industry standard for controlling modems. It allowed for performing actions such as selecting the communication protocol, setting the line speed, and dialing a number [40]. The International Telephone Union (ITU-T) codified the AT command set over the telephone network in Recommendation V.250 [35]. In the late 90s, ETSI standardized the AT command set for GSM [26] and SMS/CBS [25], and later for UMTS and LTE [27]. Based on the Hayes and GSM AT command sets, additional AT commands were introduced for CDMA and EVDO [42, 43].

Manufacturers of cellular baseband processors (which provide modem functionality in cellular devices) have added proprietary and vendor-specific AT commands to their chipsets [18, 34, 45]. As a result, smartphones also support their own AT command sets and expose modem and/or serial interfaces once connected via USB to receive these AT commands. In some cases, these vendor-specific AT commands are designed to be issued by software to invoke specific functionality, (e.g., backing up contact information on a PC). These vendor-specific commands often do not invoke any functionality related to telephony, but to access other resources on the device. Android phone makers further extended the AT command set by adding Android-specific commands (e.g., to enable debugging over USB) to be consumed by the Android OS running on the application processor [58]. These AT commands are also usually sent over a USB connection.[1]

[1] It is also possible to send a subset of AT commands over Bluetooth, although functionality is limited [21].

2.2 USB on Android

As the most important and widely adopted peripheral interface in the Android ecosystem, USB is responsible for a number of important tasks, including battery charging, data transmission, and network sharing with other devices. To accomplish these tasks via USB, Android devices support three different USB modes: host, device, and accessory mode. USB device mode, the most common mode and our focus because of its widespread use, is used when the phone connects to a PC and emulates device functionality such as exposing an MTP (Media Transfer Protocol) endpoint.

As shown in Figure 1, the Android USB implementation in device mode relies on both the Linux kernel and the Android framework. In the kernel space, the Android composite driver exposes a `sysfs` entry to user space and interfaces with the kernel's USB gadget driver. Different USB functionalities (such as USB Mass Storage or MTP) require different gadget drivers to be loaded. The gadget driver sits above the USB controller driver, which communicates with the USB device controller (UDC) hardware. Within the user space, the Android *UsbService* provides Java interfaces to applications, instantiating *UsbDeviceManager* and *UsbSettingsManager* to enable users to switch between different USB functionalities. The Android *init* daemon typically takes care of setting user-requested USB functionality by loading an `init.usb.rc` script during startup. This init script contains detailed procedures for setting functionality on the phone, essentially writing data to the `sysfs`.

2.3 Android USB Modem Interface

USB Modem functionality in Android can be accessed if the smartphone vendor exposes a USB CDC (Communication Device Class) ACM (Abstract Control Model) interface from within their phones. Once enabled and connected, this creates a `tty` device such as `/dev/ttyACM0`, enabling the phone to receive AT commands over the USB interface [47]. As shown in Figure 1, the `tty` device relays AT commands to the Android user space. Vendor-specific native daemons read from the device file, and take further actions based on the nature of the AT command. These daemons can handle vendor/carrier-added AT commands, such as "AT+USBDEBUG" (enabling USB debugging) locally, without notifying the upper layer. Otherwise, (pre-installed) applications will be triggered to process the commands. These AT commands are often designed to provide shortcuts for managing, testing, and debugging within Android. For Hayes and GSM AT commands, such as "ATD" (which enables voice dialing), the RIL (Radio Interface Layer) will be triggered to deliver the command to the baseband processor.

Vendor	# of Firmware	# of AT Commands
ASUS	210	803
Google	447	291
HTC	55	299
Huawei	83	1122
LG	150	450
Lenovo	198	1008
LineageOS	199	535
Motorola	145	779
Samsung	373	1251
Sony	128	416
ZTE	30	696
Total	2018	3500

Table 1: Per vendor counts of firmware images examined and AT commands extracted from all images.

2.4 Threat Model

Throughout the paper, we assume a malicious USB host, such as a PC or a USB charging station controlled by an adversary, tries to attack the connected Android phone via USB. We assume the attacker is able to access or switch to the possibly inactive AT interface — if available. With access to this interface, the attacker will be able to send arbitrary AT commands supported by the target device to launch attacks. We assume that all of these attacks can be fully scripted and only require a physical USB connection. Additionally, we assume that Developer Options and USB Debugging are disabled by default. During the extraction of AT commands from firmware images, we assume that the existence of AT commands in binaries and applications are not purposefully obfuscated, encrypted or compressed.

3 Design & Implementation

We design and implement methods to extract, filter, and test AT commands found within the Android ecosystem. Our procedure for acquiring these commands is shown in Figure 2. We begin by identifying and retrieving 2,018 Android binary smartphone firmware images, covering 11 major Android cellphone vendors. The details of this corpus are shown in Table 1. Next, for each firmware, we unpack the image using a variety of tools and extract AT command strings using a regular expression. After additional filtering, we recover 3,500 unique AT commands, many of which have differing parameter strings.[2] Finally, using this database, we evaluate the security impact of these commands on real Android devices by setting up an automated testing framework to send the commands to physical Android devices and monitor any side-effects.

[2] We extracted a total of 7,281 AT commands when different parameter strings are included.

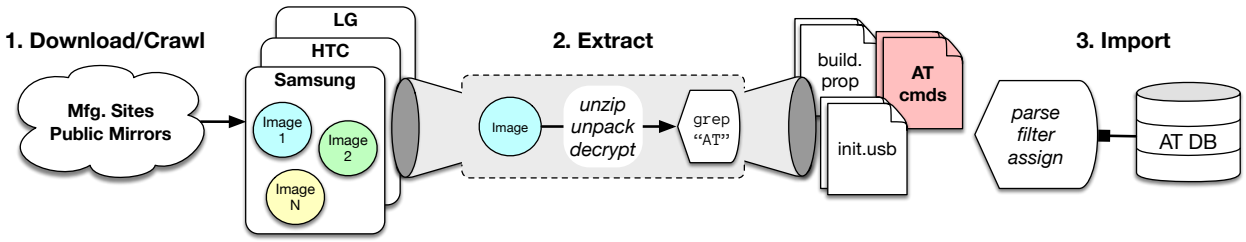


Figure 2: A graphical depiction of our paper's Android firmware image processing pipeline.

3.1 AT Command Extraction

We first gather Android firmware images from manufacturer websites and third-party hosts. For more details on the downloading process, see Section A.3. With a corpus of firmware images, we begin extraction and filtering for commands. We traverse each Android firmware image as deeply as possible, recovering unique AT commands and parameter combinations. Additionally, we also capture build information for each image from the standard Android build properties file, `build.prop`. This file provides key metadata about the image itself. We also collect any USB init/pre-configuration files (e.g., init.usb.rc) found in Android boot images to gain insight into the USB modes supported by each firmware.

In order to find AT commands present in firmware images, we look in every file for any string containing the regular expression `AT[+*!@#$%^&]`. AT commands with a symbol immediately following the ATtention string are known as *extended AT commands*. Original Equipment Manufacturers (OEMs) are free to add any amount of extended commands to their products. We focus on solely collecting AT extended command references within these firmware images for later categorization and testing.

Many pieces of firmware were archived using standard formats. Vendor-specific formats included: HTC's .exe format, unpackable using the HTC RUU Decrypt Tool [12]; Huawei's update.app format, unpackable using splitupdate [10]; LG's .kdz/.dz format, unpackable using LGE KDZ Utilities [7]; and Sony's .ftf format, unpackable using 7-Zip. Any nested archives directly under the top-level archive (e.g., Samsung images' several nested tars) are similarly unpacked.

Once all files are extracted from the archives, we process each file according to its characteristics. For native binaries, such as ELF, we are limited to using `strings` to extract all possible strings, over which we `grep` for any of our target AT prefixes. For text-based files, `grep` is applied directly to catch potential AT commands. For ZIP-like files, we unzip and traverse the directory to examine each extracted file. ZIP-like files include yaffs (unpacked using *unyaffs* [13]), Lenovo's SZB (unpacked using *szbtool* [11]) and Lenovo's QSB (unpacked using a qsb splitter [6]) formats.

If the file is a VFAT or EXT4 filesystem image (e.g., `system.img`), we mount the filesystem and traverse it once mounted to check each contained file. Filesystem images are not always readily mountable. They may be single or split-apart sparse Android images, which we first convert into EXT4 using the Android *simg2img* tool [9]. They may be provided as unsparse chunks, which need to be reconstituted according to an instruction XML file indicating start sector and number of partition sectors for each chunk. They may otherwise be provided as sparse Android data images (SDATs), which are converted into EXT4 using *sdat2img* [8]. Sony filesystem images, in particular, may be given in SIN format, which are converted into EXT4 using *FlashTool* [3].

Android filesystem partitions contain APK files, which we decompile using *dex2jar* [2] and *jd-cli* [5] treating the output as text files to pull AT commands from. Similarly, we also decompile JAR files using *jd-cli* before extracting AT commands from them. Any discovered ODEX files are first disassembled using *baksmali* [1], after which we look for AT commands in the assembly output. We then reconstruct the DEX file using the assembly output with *smali* and decompile it using *jadx* [4] before looking for AT commands in the resulting output.

3.2 Building an AT Command Database

After AT commands are extracted from each image, we develop a script to parse the "AT" matches. This script applies additional filtering with a more strict regular expression and uses a scoring heuristic to eliminate commands that appear to be invalid.

For every command found, we record metadata such as the vendor, image, and filename where it was discovered. Additionally we find any parameters to the AT command and store the unique combinations with the command. To organize the data, we use MongoDB with a single top-level document for each vendor. Each vendor has an array of images, which in turn have Android metadata, including, but not limited to, Android version, phone model, and build ID. Finally, each image has a list of AT commands.

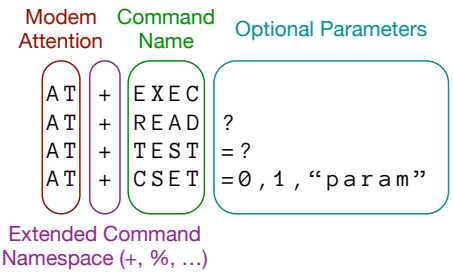


Figure 3: A colorized representation of AT command syntax.

```
1  (?:[^a-zA-Z0-9]|^)  # Left of the AT must NOT
2                      # be a letter or number
3
4  (?P<cmd>            # Capture the match
5    AT[!@#$%^&*+]     # Match AT[symbol]
6    [_A-Za-z0-9]{3,}  # Match the name and
7  )
8
9  (?P<arg>            # Capture the match
10   \?  |             # Match AT+READ?
11                     # Match AT+CSET=0,1,"param"
12   =["'+=;%,?A-Za-z0-9]+ |
13   =\? |             # Match AT+TEST=?
14   =                 # Match a blank parameter
15 )?                  # Match AT+EXEC
```

Figure 4: The regular expression developed to match extended AT commands. The regular expression syntax is from Python. All white space is ignored. Note that the regex is matching both text files and binary data.

Filtering Lines containing AT commands as discovered using `strings` and `grep` are what we call *coarse-grained matches*. This means any matching lines may be *invalid* or *spurious*. We define an invalid match to mean not conforming to the expected patterns of an AT command. Figure 3 shows the syntax of an AT command, with different colors describing the modem attention string, command delimiter, command name, and parameter string. It also shows the four primary uses of AT commands: executing an action, reading from a parameter, testing for allowed parameters, and setting a parameter. In practice, what these types *actually* do is left up to the implementation. Regardless, these four types are the standard syntax patterns we aim to match.

To capture these four types, we develop a regular expression as shown in Figure 4 to match their syntax. Line 1 of the RE will ignore any matches that are *not* at the beginning of the matched line *and* have a letter or number immediately to the left of the “AT” directive. Line 4-7 will capture and match the AT directive, the extended command namespace symbol, and the command name, which *must* be greater than or equal to three characters and only contain letters, numbers and underscores. Lines 9-15 will capture any optional argument to the command.

Specification	Usage	# of AT Commands
Hayes [16, 17]	Basic	46
ITU-T V.250 [35]	Application	61
ETSI GSM 07.05 [25]	SMS	20
ETSI TS 100 916 [26]	GSM	95
Total (unique)		222

Table 2: Additional AT commands were manually collected from several specification documents, for a total of 222 unique AT commands.

Line 10 will match a read variant, line 12 a set variant with a non-zero amount of numeric parameters, string parameters, and nested AT commands separated by semicolons (e.g., `AT+CMD=1,10,"var";+OTHER=1,2`). Line 13 will match the test variant and finally line 14 will match an empty parameter.

Despite this more restrictive regular expression, certain commands such as `AT$L2f`, `AT+_baT`, and `AT^tAT` commonly end up in the AT command database. Upon testing and visual inspection, we define commands of this appearance to be *spurious matches*. These false positive matches polluted our analytics and cause a large increase in unique commands, which in turn slows down our testing. By observing the make-up of these invalid commands, we developed a simple heuristic to score commands based off of three features: the command length, the character classes present, and the valid to invalid command ratio of the file in which it was discovered. For more details on this heuristic visit Section A.2.

In summary, the regular expression helped us discard 33.2% of all 1,392,871 processed lines across all images. The heuristic eliminated an additional 2.4% of all processed lines and brought the total unique AT command count down from 4,654 to 3,500, a 24.8% reduction. With less invalid commands matched, the signal to noise ratio of database increased and our AT command testing was faster.

Generating a DB Once we have filtered and stored every AT command along with any found parameters, we generate plain-text DB files for input into our testing framework. We create DB files containing every unique command and parameter and vendor-specific ATDB files. These give us different test profiles for phone testing. In addition, we also manually collect AT commands from multiple specifications, as shown in Table 2. Many of these commands are not extended AT commands (`AT[symbol]`) and would not be matched during our filtering step. Also, these AT commands may not be found inside the Android firmware, but should be supported by baseband processors meeting the public specifications. Thus, we include these in our database.

3.3 AT Command Testing Framework

After all command databases have been built, we are able to send AT commands to phones with an exposed AT interface. To achieve this, we developed a Python script running on Ubuntu 16.04 that uses `PySerial` to interact with the phones. When a phone that exposes an AT interface is plugged in, the Linux kernel will read its USB configuration descriptor and load any necessary drivers. To Linux, the modem interface appears as a Communication Device Class (CDC) Abstract Control Model (ACM), which causes the `usbserial` driver to be loaded. This driver creates one or more `/dev/ttyACM` device nodes. PySerial opens and interacts with these device nodes directly and sets parameters such as the baud rate and bitwidth. In practice, we were able to communicate with all modems using a 115200 baud, 8-bit, no parity, 1 stop bit scheme.

For some manufacturers, the USB modem interface is not included in the default USB configuration. In this case, there may be a second hidden configuration than can be dynamically switched to using `libusb` directly. We use a public tool called *usbswitch* [47] to select the alternative USB configuration, enabling communication over the modem interface. Once a modem is exposed, we send a command, wait for a response or a timeout, and log both sides of the conversation for future review. This logging is *crucial* for understanding what unknown commands are doing to a phone under test.

During our preliminary testing, we discovered commands that reboot, reset, or cause instabilities in the phone. We thus blacklist certain commands to allow our framework to continue without human intervention. These blacklisted commands are returned to for further manual inspection. For suspicious commands, we manually rerun them on the target phone couple of times to narrow down on the exact functionality and behavior.

4 AT Command Analysis

To understand the prevalence and security impact of AT commands on the Android ecosystem, we perform firmware analysis and runtime vulnerability analysis, and we launch attacks. In the firmware analysis, we first examine the entire corpus of AT commands extracted from firmware to discover trends in their occurrence across vendors and Android versions. Our goal is to gain insight into the general usage of AT commands from within the Android ecosystem. We then take a closer look into the native binaries and applications that contain the most AT commands per vendor. This information advises which binaries to put into IDA for further analysis. We also inspect the USB configuration files inside these images and provide an estimate of how many images may potentially expose the USB modem interface.

In the runtime vulnerability analysis, we first look at 14 Android devices to confirm their exposure of a USB modem interface. We launch our AT command testing framework on 8 different Android devices that do expose such an interface and collect command information based on both response and observable effects on the physical devices during our testing. We categorize these commands and further show their security impact. We leverage the knowledge gained of AT commands from runtime and IDA analysis to create new attacks using AT commands, and we verify these attacks on off-the-shelf Android phones.

4.1 Firmware Analysis

Distribution of AT Commands Across Vendors. We look at the number of unique AT commands across select vendors, namely Google, Samsung, and LG. As the base of all other Android variants, AOSP (Android Open Source Project) keeps the number of AT commands contained inside the factory images around 70 on average. Figure 5a shows the distribution of these commands across AOSP firmwares. The average amount of AT commands is fewer than 100 across all versions, and is under 75 starting from version 4.3. Version 4.2 has the largest variance across different images. We correlate this with the wide product line support of the Nexus series, which later became the Pixel phone series.

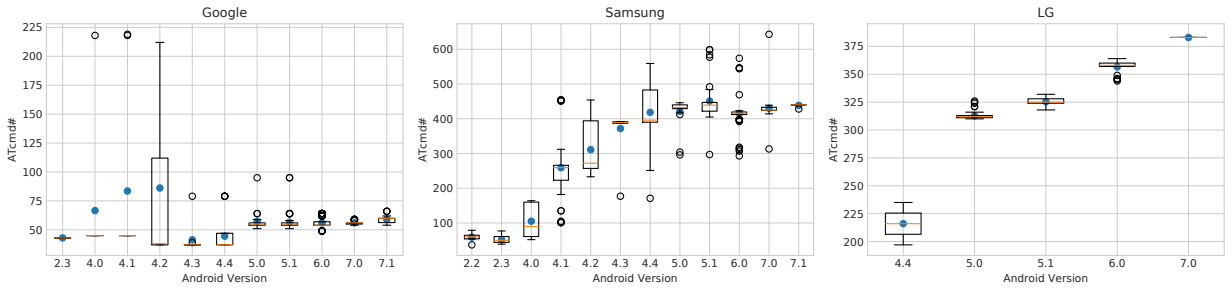
New AT commands are constantly added into stock ROMs due to vendor-specific customizations. Figure 5b presents the number of AT commands found in Samsung Android images. Our results show that the number of AT commands generally increases across different versions before Android 5.0. Although the average number stays fairly stable after version 5.0, it is still above 400. This means that given an image, Samsung has at least 300 additional AT commands compared to its AOSP counterpart. This trend is even more apparent for LG, with the number of AT commands increasing monotonically as the Android version grows, as shown in Figure 5c. The average number of AT commands within LG Android version 7.0 images is over 375.

AT Command Top 10. Table 13 in the Appendix shows the frequency of each of the top 10 most frequent AT commands overall and per different major vendor. All of the top 10 from the aggregation are standard GSM AT commands, which manage modems and calls. Similarly, all of the most frequent commands found in AOSP images are also GSM-related. In contrast, 3 non-standard AT commands ("AT+DEVCONFINFO"[3], "AT+PROF"[4], and "AT+SYNCML"[5]) are among the most common ones in

[3] Get the device configuration information.

[4] Retrieve information, such as "AT+PROF="Phonebook"".

[5] Synchronization Markup Language support for device syncing.



(a) ATcmd Distribution of Google. (b) ATcmd Distribution of Samsung. (c) ATcmd Distribution of LG.

Figure 5: AT Command distribution across three major Android smartphone manufacturers.

Google	**ATcmd#**
/vendor/lib/libsec-ril_lte.so	183
/lib/libxgold-ril.so	73
/lib/libreference-ril.so	37
/lib/hw/bluetooth.default.so	23
/lib/bluez-plugin/audio.so	19
Samsung	
/bin/at_distributor	331
/md1rom.img	226
/app/FactoryTest_CAM.apk	145
/bin/sec_atd	142
/bin/engpc	140
LG	
/bin/atd	354
/lib/libreference-ril.so	37
/lib/hw/bluetooth.default.so	27
/app/LGATCMDService/arm/LGATCMDService.odex	19
/app/LGBluetooth4/arm/LGBluetooth4.odex	15

Table 3: Top 5 binaries which contain the most AT commands per Google, Samsung, and LG.

Samsung images besides the 7 GSM-related commands. Surprisingly, 8 of the top 10 AT commands in LG are non-standard (prefixed by "AT%"). Further investigation shows them all to be vendor-specific. We extend our inspection to the top 20 AT commands and find the trend to be the same – the most frequent AT commands are standard for Google, a combination of standard and homemade for Samsung, and mainly vendor-specific for LG.

AT Command Usage Per Binary. To see where these AT commands come from, we summarize the source of these commands and show the top 5 binaries that contribute the most commands for Google, Samsung, and LG. As shown in Table 3, most of the AT commands come from the RIL in Google. Note that some Bluetooth modules also contain AT commands. For Samsung, besides the modem image (`md1rom.img`), we could find Samsung-specific native daemons, such as `at_distributor`. A factory testing app is also listed. For LG, `atd` seems to be the sole native daemon, taking care of the most AT commands. Two LG-specific apps also appear to serve some AT commands.

To gain deeper insight into how AT commands can affect these systems, we analyzed the flow of AT commands starting from the gadget serial TTY device (usually `/dev/ttyGS0`) to any native daemons and finally to other devices or system applications. We analyzed the LG G4 and the Samsung S8+ images by reading the relevant USB init scripts and any native daemons using IDA Pro 7.0. We paired this with manual testing using the AT interface while monitoring the system with `logcat`.

Samsung S8+. Samsung's heavy use of AT commands was confirmed through analysis of four key native daemons: `ddexe`, `at_distributor`, `smdexe`, and `port-bridge`. The "Data Distributor" `ddexe` opens the primary `/dev/ttyGS0` device, monitors USB for state changes, creates a UNIX domain socket server, and routes TTY data to clients. `at_distributor` connects via UNIX socket (`/data/.socket_stream`), receives commands, and either handles them itself or dispatches them to appropriate parts of the system.

As a result of previous work (CVE-2016-4030, CVE-2016-4031, and CVE-2016-4032), Samsung has locked down the exposed AT interface with a command whitelist. This whitelist is active when the `ro.product_ship` property is set to true and limits the commands to information gathering only. Any non-whitelisted command responds with the generic reply of `OK`, even if it is invalid.

LG G4. LG follows a similar structure to handling AT commands. Its primary daemon `atd` reads and writes to the gadget serial TTY device and handles or bypasses AT commands. Some commands are handled by a static dispatch table within `atd` and may propagate throughout the system via UNIX domain socket `/dev/socket/atd`. *LGATCMDService* is an Android background service that listens for and handles any incoming commands before sending back a response. At least 89 different commands

Vendor	USB.rc w/ `acm`	Avg. `acm` Triggers	USB.rc w/ `diag`	Avg. `diag` Triggers
ASUS	330	2.9	262	2.5
Google	73	5.6	496	29.2
HTC	253	14.3	253	31.3
Huawei	56	5	58	29.1
Lenovo	144	6.7	100	25.7
LG	591	1.1	693	1.0
LineageOS	168	4.4	281	15.1
Motorola	10	16	224	7.0
Samsung	581	5.4	509	19.6
Sony	56	4.7	56	21.2
ZTE	23	6.9	23	36.5
Total	2955	4.1	2285	17.3

Table 4: Per vendor counts of USB.rc files found to contain `acm` and `diag` triggers, alongside the average number of contained triggers. In total, we found 12,018 `acm` and 39,605 `diag` triggers across USB.rc files in 1,564 images.

Device	Android Ver#	Modem Exposed
Samsung Galaxy Note 2	4.4.2	Y
Samsung Galaxy S7 Edge	7.0	Y
Samsung Galaxy S8 Plus	7.0	Y
LG G3	6.0	Y
LG G4	6.0	Y
HTC One	4.4.2	Y*
HTC Desire 626	5.1	N
Asus ZenPhone 2	5.0	Y (root)
Asus ZenPad	5.0.2	Y (root)
Google Nexus 5	5.1.1	Y (root)
Google Nexus 5X	6.0	Y (root)
Google Nexus 6P	7.1.1	N*
Google Pixel	7.1.1	N
Motorola Moto X	5.1	N*

Table 5: We examined 14 Android devices to find if they expose USB modem interfaces. 6 expose the modem by default; 4 can expose it after being rooted.

are handled by this application and, given its extensive system permissions, it is an interesting target. A previous vulnerability in 2016 [49] (CVE-2016-3117) gave any application the ability to communicate through *LGATCMDService* to `atd`, allowing the phone to be bricked or sensitive data to be read. Through static analysis of this service APK, we confirmed that there were now checks ensuring that only requests from the `system` user (UID 1000) would be allowed. Despite this patch, unlike Samsung, LG does not whitelist AT commands, so any that are supported by the Android system or modem are passed through the USB interface.

USB Pre-Configuration Files. Now that we know the prevalence of AT commands in the gathered firmware images, we inspect the susceptibility of the images to AT commands. We do this by looking at USB init/pre-configuration files (e.g., `init.usb.rc`), referred from here on as USB.rc files, which provide details about the USB modes supported by the device. We were able to extract pre-configuration files from 1,564 of the 2,018 total images, some having multiple such files (for example, HTC images contain an average of 10).

We look for `property:sys.usb.config` triggers in the pre-configuration files and discover that 864 images (55% of the images from which USB.rc files were successfully extracted) contain at least one USB.rc file with triggers for ACM mode. Since enabling USB modem functionality causes a CDC-ACM interface to be exposed, our finding suggests that over half[6] of phone firmwares have the potential to provide modem functionality. We also look for triggers for diagnostic mode, indicated by `diag`, which usually activated the ACM interface once enabled. We discover that 1,175 images (75% of the images from which USB.rc files were extracted) contain at least one USB.rc file with `diag` triggers. Our finding suggests that even more phone firmwares (beyond those with ACM mode triggers) have the potential to provide modem functionality through alternative `diag` triggers.

Table 4 presents the breakdown of average `acm` and `diag` trigger counts per vendor. Since each image may have multiple USB.rc files, we average trigger counts over the total number of these files from each vendor, rather than the number of images containing USB.rc files.

4.2 Runtime Vulnerability Analysis

We first examine the prevalence of the USB modem interface being exposed by different Android devices. We look at 13 Android phones and 1 Android tablet from major vendors. Table 5 provides an overview of these devices and whether or not they expose a modem interface. All Samsung and LG phones we tested expose a USB modem interface by default. HTC One also exposes a modem interface, but it does not accept any AT commands. ZenPhone 2, ZenPad, and Nexus 5/5X also expose a modem interface, but not by default; their USB configuration must be changed after rooting. Of note, Zenpad, though it does not support mobile data at all, still exposed a modem interface. Although neither Nexus 6P nor Moto X reveal a modem interface during our testing, they have the potential to enable a modem interface by exploiting fastboot vulnerabilities [31].

We chose 8 devices shown in Table 6 for testing, including all devices exposing a USB modem interface by default, as well as 3 other devices that offer ways to enable such an interface. We use our AT command testing framework to send the 3500 unique AT commands we ex-

[6] We expect a similar prevalence of ACM mode triggers among the remaining 454 images for which extraction of USB.rc files failed.

Device	Build Number	USB Config
Note2	KOT49H.N7100XXSFQA7	None
S7Edge	NRD90M.G935FXXU1DQB7	None
S8+	NRD90M.G955USQU1AQD9	None
G3	MRA58K	None
G4	MRA58K	None
ZenPhone2*	LRX21V.WW-ASUS_Z00A-2.20.40.198_20160930_875_user	system.at-proxy.mode [1-4]
ZenPad*	LRX22G.WW_ZenPad-12.26.4.69-20170410	sys.usb.config mtp,acm
Nexus5*	LMY48I	sys.usb.config diag,adb

Table 6: We chose 8 devices from Table 5, including 5 phones exposing the modem by default, and 3 rooted devices (as marked by *) with the modem exposed by setting the USB configuration. We tested all of them using our AT command testing framework.

Command	Action	Tested Phones
AT%RESTART	Phone reboot	G3
AT%PMRST	Phone reboot	G3
AT%POWEROFF	Phone reboot	G3/G4
AT%DLOAD	Firmware download mode	G3/G4
AT%FRST	Factory reset	G3
AT%MODEMRESET	Modem reset	G3/G4
AT+CRST=FS	Factory reset	G3/G4
AT+CFUN=0	Phone Reboot	G3/G4
AT+CFUN=1,1	Phone reboot	S7Edge/S8+
AT+CFUN=1,1	MiniOS and factory reset status 2	G4
AT+CFUN=6	Phone reboot	G3/G4/S8+
AT+CFUN=6,0	Phone reboot	S8+
AT+FACTORST=0,0	Factory reset	S7Edge/S8+
AT+SUDDLMOD=0,0	Firmware download mode	Note2/S7Edge/S8+
AT+FUS?	Firmware download mode	Note2/S7Edge/S8+
AT^RESET	Phone power off	G3/G4/S8+

Table 7: A selection of commands that can change the phone's firmware image through resetting or updating.

tracted, plus 222 standard commands, to each device. We manually look at the response elicited for each command, picking up the ones with successful replies or observable side effects during testing, e.g., causing the device to reboot. We are able to group notable behaviors into several categories that demonstrate the wide security impact of AT commands using this USB modem interface, which is either exposed by default or enabled later by other means, e.g., by rooting the device.

4.2.1 Firmware Flashing

We find AT commands enabling firmware flashing in Android phones, which were reported before [20]. Once the phone is put into download mode using the AT commands in Table 7, attackers can attempt to flash rooted or malware pre-installed images into the phone. On the Samsung phones we tested, the AT commands put the phone into Odin [48] mode, although they were not able to bypass the device standard firmware authentication mechanism [57, 30]. Odin also sets the KNOX warranty fuse within a phone if an unsigned firmware image is flashed. We also found LG has its own firmware flashing AT command, shown in Table 7, which allows flashing malicious firmware into the phone using LGUP [39].[7] Factory resetting AT commands are also found, erasing user data without permission. Some commands reboot/shutdown the phone, and we manually inspect security related settings, e.g., USB debugging, after the reboot, but did not find any particular configuration change.

Command	Action	Tested Phones
ATD	Dial a number	G3/G4/S8+/Nexus5/ ZenPhone2
ATH	Hangup call	G3/G4/S8+/Nexus5/ ZenPhone2
ATA	Answer incoming call	G3/G4/Nexus5
AT%IMEI=[param]	Allows the IMEI to be changed	G3/G4
AT%USB=adb	Enables invisible ADB debugging	G3/G4
AT%KEYLOCK=0	Unlock the screen	G3/G4
AT+CKPD	Sends keypad keys ([0-9*#])	G3/G4/S8+
AT+CMGS	Sends a SMS message	ZenPhone2
AT+CGDATA	Connect to the Internet using data	G3/G4/Nexus5/ ZenPhone2
AT+CPIN	SIM PIN management	G3/G4/S8+/Nexus5/ ZenPhone2
AT$QCMGD	Delete messages (by index, all read/sent)	Nexus5

Table 8: A selection of commands that can be used to gain further access into the phone.

We observe that some AT commands result in different behaviors on phones from different vendors. As an example, "AT+CFUN=1,1," although a standard command that is supposed to "reset the device and provide full functionalities"[8] according to the GSM spec [26], causes Samsung phones to reboot and causes LG G4 to become bricked and show "LG G4 factory reset status 2" blue screen error. Surprisingly, the USB modem interface was still exposed even in this mode. While we were unable to restore the phone using any of the normal procedures, we were able to successfully un-brick the phone using a combination of "AT%MODEMRESET", which changes the factory reset status from 2 into 5, and "AT%RESTART" commands, which finally reboots the phone into a normal booting environment following a factory reset.

4.2.2 Android Security Bypassing

This section demonstrates AT commands that bypass different Android security mechanisms, such as lock screen, UI notification, etc., as shown in Table 8. We were able to make phone calls by sending an "ATD" command to the phone. Note that this command works even if there is a screen lock on the phone. Combined with "ATH" and "ATA," one can call any number, accept any incoming call, and end a call via a USB connection. Note that the ATD vulnerability on Samsung phones was reported 2 years ago [47], and it was patched. Neither our Note 2 nor S7 Edge is able to make a call. Nevertheless, this once-patched vulnerability reappears on the S8+. Similarly, AT commands for managing PINs on SIM cards and connecting to the Internet using mobile data were also accessible on four of the testing phones. These commands are all standard AT commands delivered to the modem by native daemons, bypassing the Android framework. We also find an LG-specific command that allows us to change the IMEI, again bypassing the Android layer.

One USB debugging enabling command is found in LG phones, together with an AT command to unlock the screen. After USB debugging is enabled using this AT command, there is no indication on the UI showing USB debugging was enabled, and there is no prompt from the UI asking for the key to be added. This shows that the whole Android layer is bypassed without being notified when we enable USB debugging using this AT command. Commands for sending touchscreen events and keystrokes are also discovered for LG phones and the S8+; we can see the indications on the screen. We suspect these AT commands were mainly designed for UI automation testing, since they mimic human interactions. Unfortunately, they also enable more complicated attacks which only requires a USB connection, as we will show in a later section.

4.2.3 Sensitive Information Leaking

While Android security has been improving over the years with respect to protecting privacy information, we found quite a few AT commands providing different kinds of information, including IMEI, battery level, phone model, serial number, manufacturer, filesystem partition information, software version, Android version, hardware version, SIM card details, etc., as shown in Table 9.[10] Vendors also introduce their own commands to ease querying. These are unauthenticated commands that can be accessed by anyone. One example command is "AT+DEVCONINFO" from S8+, providing detailed information about the phone as shown in Figure 6. Shown in bold are examples of sensitive device information, including device model (MN), serial number (SN), IMEI, and connection over MTP.

Command	Action	Tested Phones
ATI	Manufacturer, model, revision, SVN, IMEI	G4/S8+/Nexus5/ ZenPhone2
AT%SYSCAT	Read and return data from `/sys/*`[9]	G3/G4
AT%PROCCAT	Read and return data from `/proc/*`	G3/G4
AT+DEVCONINFO	Phone model, serial number, IMEI, and etc.	Note2/S7Edge/S8+
AT+GMR	Phone model	G3/G4/Note2/S8+/ ZenPhone2
AT+IMEINUM	IMEI number	Note2/S7Edge/S8+
AT+SERIALNO	Serial number	Note2/S7Edge/S8+
AT+SIZECHECK	Filesystem partition information	Note2/S7Edge/S8+
AT+VERSNAME	Android version	S7Edge/S8+
AT+CLAC	List all supported AT commands	G3/G4/S7Edge/Nexus5/ ZenPad/ZenPhone2
AT+ICCID	Sim card ICCID	G3/G4/Nexus5

Table 9: A selection of commands that leak sensitive information about the device.

```
[['AT+DEVCONINFO\r+DEVCONINFO:
MN(SM-G955U);BASE(SM-N900);VER(G955USQU1AQD9/
G955UOYN1AQD9/G955USQU1AQD9/G955USQU1AQD9);
HIDVER(G955USQU1AQD9/G955UOYN1AQD9/G955USQU1AQD9/
G955USQU1AQD9);MNC();MCC();PRD(VZW);;OMCCODE();
SN(R38HC09NWMR);IMEI(354003080061555);
UN(9887BC45395656444F);PN();CON(AT,MTP);LOCK(NONE);
LIMIT(FALSE);SDP(RUNTIME);HVID(Data:196609,
Cache:262145,System:327681);USER(OWNER)\r',
'#OK#\r', 'OK\r']]
```

Figure 6: Output from "AT+DEVCONINFO" on a Samsung S8+. Note information in bold corresponding to model number, serial number, IMEI, and connection type.

We also find 3 AT commands that report all supported AT commands on the device. "AT+CLAC" is a standard command; "AT+LIST" only works on Nexus 5; and "AT$QCCLAC" appears to be a Qualcomm-specific command supported by Qualcomm baseband processors. Note that both "AT+CLAC" and "AT$QCCLAC" could be supported at the same time within a device, returning different lists of supported AT commands. We also leveraged these commands to limit the scope of AT commands to try when we attempted to un-brick the LG G4.

4.2.4 Modem AT Proxy

Unlike other Android devices, which rely on `sys.usb.config` to manage the USB functionality, ASUS ZenPhone 2 has a unique setting to enable the

[7] While Odin wipes everything by default, LGUP leaves the user data intact in the device if "Upgrade" mode is chosen.

[8] Level "full functionality" is where the highest level of power is drawn.

[9] We discovered a bug leading to arbitrary file reads in the AT%PROCCAT and AT%SYSCAT commands. See Section 4.3 for more details.

[10] For more such commands, please refer to Table 14 in the Appendix.

Command	Action	Tested Phones
AT+XDBGCONF	Debug configuration	ZenPhone2-mode2/ ZenPad
AT+XABBTRACE	BB trace configuration	ZenPhone-mode2/ ZenPad
AT+XSYSTRACE	System trace port configuration	ZenPhone2-mode2/ ZenPad
AT+XSIMSTATE	SIM and phone lock status	ZenPhone2-mode2/ ZenPad
AT+XLOG=95,1	Exception log reading	ZenPhone2-mode2/ ZenPad
AT+XLEMA	Emergency number reset	ZenPhone2-mode2/ ZenPad
AT+XNVMPLN	PLMN info list for GSM, UMTS, and LTE tables	ZenPhone2-mode2

Table 10: Commands specific to the AT proxy mode that allows debugging and tracing inside the modem.

hidden modem interface, called AT proxy mode, as shown in Table 6. This modem AT proxy does not appear to be specific to ASUS, but also occurs on Android devices running Intel Atom processors from other vendors, including Intel itself. According to Intel, "this functionality provides the link to Modem to allow to use AT commands through a virtual com port" [33]. Many commands found in ZenPhone 2 also work on ZenPad.

There are 4 modes in ZenPhone 2, numbered from 1 to 4. Based on our testing, mode 1 works like a traditional modem and responds to most of the AT commands from the standards, including making a call using "ATD" and sending a SMS message using "AT+CMGS". While most standard commands still work on mode 2, a new series of command starting with "AT+X" are also supported. We list some of these in Table 10. We base our detailed description for each command on online documentation from Telit [51]. Mode 3 is similar to mode 2, except for truncation of responses to some commands. Some commands stop working as well in mode 3, e.g., "AT+XABBTRACE". Mode 4 is similar to mode 3, except more commands worked without returning errors, such as "AT+GMI" and "AT+GMM". In general, once this AT proxy mode is enabled, attackers can exfiltrate any information within the modem by setting debug or trace points.

4.2.5 Others

We present other commands which do not directly fit into the previous categorizations but have security impacts as well in Table 11. For example, we found 3 hidden menus on LG phones during our testing, including MiniOS menu, Hidden menu[11], and MID result menu. All of them provide different information, testing, and debugging functionalities. Even though these hidden menus were exploited before [22], they still exist and can be triggered by a single AT command. Interestingly, the command used to trigger the hidden menu is also found on Nexus 5. We suspect that it is partially because Nexus 5 was made by LG. However, the response of the command is overwritten to return the baseband version. Instead, a separate AT command was added into Nexus 5 to provide engineering debugging information.

[11] It is called Hidden menu.

Command	Action	Tested Phones
AT+VZWAPNE	Manage APN settings	G3/G4
AT$SPDEBUG	Engineering debugging information	Nexus5
AT%MINIOS	MiniOS mode	G3/G4
AT%VZWHM	Hidden menu	G3/G4
AT%VZWHM	Baseband version	Nexus5
AT%VZWIOTHM	Baseband version	Nexus5
AT%AUTOUITEST	MID result menu	G3/G4

Table 11: A section of commands that provide APN settings, debugging information, and hidden menus.

4.3 Attacks

After analyzing many AT commands across vendors, we have narrowed down the set to a selection of useful or interesting commands from an attacker's perspective. To demonstrate the potential impact of exposed AT interfaces on phones, we describe actual and theoretical attacks that may be mounted against them.

Lockscreen Bypassing & Event Injection. With the discovery of the LG G4's AT interface and knowledge of certain AT commands, we developed a proof of concept attack against the phone in order to enable USB debugging without any user interaction. Access to USB debugging and developer options would allow a local attacker connected to USB to install new unsigned applications with high permissions to achieve persistence on a victim's phone. Additionally, they may be able to further compromise the system using an Android Kernel exploit through the Android Debug Bridge (ADB).

To demonstrate this attack, we combine AT commands to (1) bypass the lock screen (`AT%KEYLOCK=0`), (2) navigate to the settings menu using touchscreen automation, and (3) allow USB debugging from our attacking machine (`AT%USB=adb`). The `KEYLOCK` AT command bypasses the lock screen even if a pattern or passcode is set [23]. From there, arbitrary touch events can be sent to control the phone.[12] Given that nearly 28% of users do not have a pin, pattern, or biometric lock [19], this attack would still be feasible even without the LG-specific `KEYLOCK` command.

[12] Once these commands are patched, visit `https://github.com/FICS/atcmd` for an automated script and the required utilities.

Confused Deputy Path Traversal. Through manual auditing of the LG G4's firmware image in IDA Pro (specifically in `atd`), we discovered that the `AT%PROCCAT` and `AT%SYSCAT` commands are intended to open, read, and send back the contents of a file relative to the `/proc` and `/sys` directories respectively. While this information alone would be useful for an attacker mounting an attack against the system, we discovered that these commands are vulnerable to a path traversal attack. This means we can send `AT%PROCCAT=../arbitrary/file` and receive back the entire `file` contents over the AT interface. *As a result, we are able to access all data in /sdcard, including arbitrary private information.* If pictures or application data is stored in the /sdcard directory, then they can be read by this attack. In addition, we attempted to access Android user data in the `/data/data/com.target.app` directories, but were unsuccessful due as no allow rule was made for `atd` to access this region. The `atd` daemon runs as the Android System user and acts within a reasonably privileged SEAndroid context. It is unclear how permissive the AT distributors' policies are, but future auditing will focus on this area.

Memory Corruption. During our manual AT command testing, we discovered multiple buffer overflows in the LG G3 & G4 `atd` process and one in the Samsung S8+ `connfwexe` daemon. Upon inspection of the tombstones (Android's crash dump), all appeared to be crashes via SIGABRT trigged from `FORTIFY` failures [36]. Although these out-of-bounds writes were caught by `FORTIFY_SOURCE` and are not exploitable, it is possible that further stress testing and auditing of these native daemons could yield an exploitable vulnerability. Given that these interfaces are undocumented and proprietary, we believe it to be unlikely that they have received audit from an external source.

If an exploitable memory corruption or Use-After-Free vulnerability were discovered on LG's system daemons, we could dynamically gather Return Oriented Programming (ROP) gadgets by using a call to `AT%PROCCAT=[pid]/exe` to leak the entire binary image and reveal Address Space Layout Randomization (ASLR) slides using `AT%PROCCAT=[pid]/maps` to get all of the memory region address ranges.

5 Discussion

Methodology Limitations. The design of the regular expression is a tradeoff between discovering as many AT commands as possible and keeping the false positive rate low. Nevertheless, we might miss some AT commands due to regex mismatching. For instance, we assume the prefix "AT" is in the capital case, and ignore the small case "at". Because "at" introduced more false alarms, and the prefix should be case insensitive according the standards. However, we did find few commands only working in the small case on certain device. Due to the limitation of static analysis, we also could not find AT commands which are built dynamically during the runtime. While our testing framework is able to send out AT commands and record response in the logging automatically, fully automated testing is still infeasible. A response may be as simple as "OK" and the side effect of a command (e.g., warning of configuration changes) might be transient. To figure out the exact impact of a command, we need to enable logcat from ADB to inspect the propagation path of the command, and stare at the phone screen during the command runtime looking for Android UI notifications. Some commands also reset the USB connection which requires human intervention to resume the testing.

USB Attack Surface. During our static and dynamic analysis, we realized that there is a lot of extra functionality hidden in phone configurations (e.g., init.usb.rc) such as DIAG (Diagnostic), DM (Diagnostic Mode), TTY/SERIAL (Terminal), SMD (Shared Memory Device), RMNET (Remote Network). This diverse functionality is a benefit of Android's mature USB gadget driver, but unfortunately compared to MTP, Mass Storage, and ADB, these USB classes are less understood or even proprietary.

These gadget interfaces all have different security implications for the phones that expose them. Depending on the protocols, they may be abused to compromise the integrity of the phone if inadvertently exposed in production. Some protocols such as DIAG offer full system control as a *feature*. This mode should *never* be exposed during a production build. Our work has shown that even access to a CDC ACM interface to input AT commands can lead to unintended information loss or act as a starting point for more sophisticated attacks. *We thus strongly recommend that manufacturers apply appropriate access controls to all debug interfaces, or disable them outright, when shipping production devices.*

"Charge-Only" Mode Effectiveness. One may expect Android's "charge-only" mode to protect against commands sent over USB, but the real-world case is more complicated. The first issue is that not every Android version supports the charge only mode. For instance, Samsung Note 2, running Android 4.4.2, does not have this option at all. Second, charge only may not be the default option when the phone connected via USB. All three Samsung phones we tested start in MTP mode by default when connected with the host machine. This enables attackers to switch to the modem interface and launch AT commands as soon as the phone is connected.

LG does better since the default option is charge only. However, once another USB option, e.g., MTP, is chosen, this option becomes the default option across reboots. With MTP enabled by default, an Android security pop-up will initially show asking to allow the host machine to access the device. But despite no choice having been made, it is already too late as AT commands may be sent to the phone immediately before Allow or Deny is chosen, effectively disabling charge only mode using `AT%USB=adb` until the next reboot. Finally, some phones may not disable all USB data even in charging mode. The Samsung S7 Edge we tested exposes the USB modem interface even after being put in charge only mode.

SELinux Effectiveness. Given the diverse and powerful functionality provided by AT commands, we wonder if SELinux could help mitigate the impact, such as preventing attackers from flashing malicious firmware into the device using AT commands. SELinux was introduced into the Android ecosystem from Android 4.3, and then became the default configurations in later versions. All the devices we tested have SELinux enabled in enforcing mode. We also did not find any AT command, which can disable or bypass SELinux.

When analyzing the LG G4 phone we discovered that its primary AT distributor daemon possessed the Linux Capabilities `CAP_SYS_ADMIN`, `CAP_DAC_OVERRIDE`, and `CAP_CHOWN`. Normally a non-root process with these capabilities would have little trouble escalating root due to the vast permissions given. With this assumption we attempted to read Android app user data using the distributor's permissions (see Section 4.3), but were blocked by SELinux's Mandatory Access Control (MAC) policy. In this case, SELinux prevented sensitive information from being leaked, but without a full audit of the policy, a bypass could still exist.

6 Related Work

The Android community has been aware of the impact of vendor customization on Android images. Felt et al. [28, 29] investigated over 900 Android applications and discovered occurrences of over privilege and permission re-delegation. Wu et al. [56] showed that 85.78% of the pre-loaded apps in 10 stock Android images are over privileged due to vendor customizations. Aafer et al. [14] analyzed the threat of hanging attribute references within pre-installed apps by looking into 97 factory images covering major Android vendors. Previous research mainly focused on apps inside the Android image, so the number of images covered was usually limited. Zhou et al. [59] studied the vulnerabilities of Linux device drivers in Android customizations, and found common issues shared by 1290 of 2423 factory images. Aafer et al. [15] discovered inconsistent security configurations among 591 custom images. Unlike previous work oriented around static analysis, we consider both static and dynamic analysis.

Communicating with the modem within a Samsung S2 using AT commands was previously detailed on the XDA forums [58]. Pereira et al. showed how to use two AT commands to flash a malicious image onto Samsung phones [46, 20]. Roberto and Aristide found additional commands working on Samsung Galaxy S4 and S6 with certain image builds [47]. Bluebug [38] showed how to exploit a security loophole within Bluetooth to issue AT commands via a covert channel to vulnerable phones. Hay discovered around 10 AT commands with security impacts on Nexus 6P due to the exposure of the AT interface exploited from the Android bootloader (ABOOT) [31]. Mickey et al. also demonstrated how to exploit the modem in cars using AT commands via USB connections [41]. Unlike previous work which focused on a single brand/-model, limited the number of AT commands covered, or rediscovered the traditional AT commands for real modems, we provide a systematic study of traditional and Android-specific AT commands in Android ecosystems across different major vendors and phone models.

While USB security has been evaluated in traditional computing environments [44, 52, 53, 32], it has received limited attention on mobile computing platforms. Stavrou et al. demonstrated how a malicious host machine can unlock the bootloader and flash a compromised system image onto an Android device using `fastboot` and `adb` via USB [55, 50]. MACTANS [37] augmented USB chargers with USB host functionalities, allowing the injection of malware into iOS devices during charging. Vidas et al. summarized Android attacks via USB [54], although the focus is mainly limited to `adb`. Due to OEM vulnerabilities in fastboot implementations, Hay also showed that hidden USB functionalities can be enabled, including modem diagnostics and AT interfaces [31], allowing data exfiltration and system downgrading.

7 Conclusion

AT commands have become an integral part of the Android ecosystem, yet the extent of their functionality is unclear and poorly documented. In this paper, we systematically retrieve and extract AT commands from over 2,000 Android smartphone firmware images across 11 vendors to build a database of 3,500 commands. We test this AT command corpus against 8 Android devices from 4 vendors via USB connections. We find different attacks using AT commands, including firmware flashing, Android security mechanism bypassing by making calls via USB, unlocking screens, injecting touch events, exfiltrating sensitive data, etc. We demonstrate that the AT com-

mand interface contains an alarming amount of unconstrained functionality and represents a broad attack surface on Android devices.

Disclosure *We have notified each vendor of any relevant findings and have worked with their security team to address the issues.*

Acknowledgments

This work was supported by the National Science Foundation under grants CNS-1540217, CNS-1526718, CNS-1564140, and CNS-1617474.

References

[1] baksmali. https://bitbucket.org/JesusFreke/smali. Last Accessed: Feb. 2018.

[2] dex2jar. https://github.com/pxb1988/dex2jar. Last Accessed: Feb. 2018.

[3] FlashTool. http://www.flashtool.net. Last Accessed: Feb. 2018.

[4] jadx. https://github.com/skylot/jadx. Last Accessed: Feb. 2018.

[5] jd-cmd - Command line Java Decompiler. https://github.com/kwart/jd-cmd. Last Accessed: Feb. 2018.

[6] Lenovo QSB File splitter. https://forum.xda-developers.com/showthread.php?t=2595269. Last Accessed: Feb. 2018.

[7] LGE KDZ Utilities. https://github.com/ehem/kdztools. Last Accessed: Feb. 2018.

[8] sdat2img. https://github.com/xpirt/sdat2img. Last Accessed: Feb. 2018.

[9] simg2img. https://github.com/anestisb/android-simg2img. Last Accessed: Feb. 2018.

[10] splitupdate. https://github.com/jenkins-84/split_updata.pl. Last Accessed: Feb. 2018.

[11] szbtool. https://github.com/yuanguo8/szbtool. Last Accessed: Feb. 2018.

[12] Universal HTC RUU/ROM Decryption Tool 3.6.8. https://forum.xda-developers.com/chef-central/android/tool-universal-htc-ruu-rom-decryption-t3382928. Last Accessed: Feb. 2018.

[13] unyaffs. https://github.com/ehlers/unyaffs. Last Accessed: Feb. 2018.

[14] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace. Hare Hunting in the Wild Android: A Study on the Threat of Hanging Attribute References. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1248–1259. ACM, 2015.

[15] Y. Aafer, X. Zhang, and W. Du. Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis. In *USENIX Security Symposium*, pages 1153–1168, 2016.

[16] ActiveXperts Software. Basic Hayes AT Command Set. https://www.activexperts.com/sms-component/at/basic/, 2018.

[17] ActiveXperts Software. Extended AT Command Set. https://www.activexperts.com/sms-component/at/extended/, 2018.

[18] ActiveXperts Software. Proprietary Sony Ericsson AT Command Set. https://www.activexperts.com/sms-component/at/sonyericsson/, 2018.

[19] M. Anderson and K. Olmstead. Many smartphone owners don't take steps to secure their devices, Mar. 2017. Pew Research Center.

[20] P. André, C. Manuel Eduardo, and B. Pedro. Charge your device with the latest malware. *BlackHat Europe*, 2014.

[21] Burak Alakus. TO CONTROL YOUR MOBILE PHONE BY AT COMMANDS VIA BLUETOOTH (C#.NET). https://burakalakusen.wordpress.com/2011/07/27/to-control-your-mobile-phone-by-at-commands-via-bluetooth/, 2011.

[22] CVE. CVE-2013-3666. https://www.cvedetails.com/cve/CVE-2013-3666/, 2013.

[23] O. Davydov. Unlocking The Screen of an LG Android Smartphone with AT Modem Commands, Feb. 2017. Forensic Focus Blog.

[24] F. Durda IV. The AT Command Set Reference - History. https://nemesis.lonestar.org, 2004.

[25] ETSI. Digital cellular telecommunications system (Phase 2+); Use of Data Terminal Equipment - Data Circuit terminating; Equipment (DTE - DCE) interface for Short Message Service (SMS) and Cell Broadcast Service (CBS) (GSM 07.05 version 5.5.0). http://www.etsi.org/deliver/etsi_gts/07/0705/05.03.00_60/gsmts_0705v050300p.pdf, 1997.

[26] ETSI. Digital cellular telecommunications system (Phase 2+); AT Command set for GSM Mobile Equipment (ME) (3GPP TS 07.07 version 7.8.0 Release 1998). http://www.etsi.org/deliver/etsi_ts/100900_100999/100916/07.08.00_60/ts_100916v070800p.pdf, 2003.

[27] ETSI. Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); LTE; AT command set for User Equipment (UE) (3GPP TS 27.007 version 13.6.0 Release 13). http://www.etsi.org/deliver/etsi_ts/127000_127099/127007/13.06.00_60/ts_127007v130600p.pdf, 2017.

[28] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[29] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, volume 30, 2011.

[30] A. Ganti. Latest Samsung Galaxy Note 8 Bootloader Prevents Flashing Unsigned Firmware on Device. https://wccftech.com/latest-samsung-galaxy-s8-s8-note8-bootloader-prevents-flashing-new-firmware/, 2018.

[31] R. Hay. fastboot OEM vuln: Android Bootloader Vulnerabilities in Vendor Customizations. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. USENIX Association, 2017.

[32] G. Hernandez, F. Fowze, D. J. Tian, T. Yavuz, and K. Butler. FirmUSB: Vetting USB Device Firmware using Domain Informed Symbolic Execution. In *24th ACM Conference on Computer and Communications Security (CCS'17)*, Dallas, USA, 2017.

[33] Intel. Installation instructions for the intel® usb driver for android* devices. https://software.intel.com/en-us/android/articles/installation-instructions-for-intel-android-usb-driver, 2015.

[34] IPFS. Motorola phone AT commands. https://ipfs.io/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco/wiki/Motorola_phone_AT_commands.html, 2014.

[35] ITU-T. Serial asynchronous automatic dialling and control. https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-V.250-200307-I!!PDF-E&type=items, 2003.

[36] J. Jelinek, A. Van de Ven, U. Drepper, and D. Novillo. Object size checking to prevent (some) buffer overflows, Sept. 2004. GCC Patches List.

[37] B. Lau, Y. Jang, C. Song, T. Wang, P. Chung, and P. Royal. Mactans: Injecting Malware into iOS Devices via Malicious Chargers. *Proceedings of the Black Hat USA Briefings, Las Vegas, NV, August 2013*, 2013.

[38] A. Laurie, M. Holtmann, and M. Herfurt. The bluebug. *AL Digital Ltd. http://trifinite. org/trifinite_stuff_bluebug. html*.

[39] LG. LGUP. https://www.mylgphones.com/download-lg-up-software, 2017.

[40] Messagestick. TECHNICAL REFERENCE FOR HAYES MODEMS. http://www.messagestick.net/modem/hayes_modem.html, 1992.

[41] S. Mickey, M. Jesse, and B. Oleksandr. Driving down the rabbit hole. In *DEF CON 25*, 2017.

[42] MultiTech Systems. AT Commands For CDMA Wireless Modems. http://www.canarysystems.com/nsupport/CDMA_AT_Commands.pdf, 2004.

[43] MultiTech Systems. EV-DO and CDMA AT Commands Reference Guide. https://www.multitech.com/documents/publications/manuals/s000546.pdf, 2015.

[44] K. Nohl and J. Lell. BadUSB-On accessories that turn evil. *Black Hat USA*, 2014.

[45] Openmoko. Neo 1973 and Neo FreeRunner GSM modem. http://wiki.openmoko.org/wiki/Neo_1973_and_Neo_FreeRunner_gsm_modem, 2012.

[46] A. Pereira, M. Correia, and P. Brandão. USB Connection Vulnerabilities on Android Smartphones: Default and Vendors' Customizations. In *IFIP International Conference on Communications and Multimedia Security*, pages 19–32. Springer, 2014.

[47] P. Roberto and F. Aristide. Modem interface exposed via USB. https://github.com/ud2/advisories/tree/master/android/samsung/nocve-2016-0004, 2016.

[48] Samsung. Samsung Odin. https://samsungodin.com/, 2017.

[49] Y. Shao, J. Ott, Y. J. Jia, Z. Qian, and Z. M. Mao. The misuse of android unix domain sockets and security implications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 80–91. ACM, 2016.

[50] A. Stavrou and Z. Wang. Exploiting Smart-Phone USB Connectivity For Fun And Profit. *BlackHat DC*, 2011.

[51] Telit wireless solutions. xN930 AT Command Reference Guide. http://www.iot.com.tr/uploads/pdf/Telit_xN930_AT_Commands_Reference_Guide_r1.pdf, 2013.

[52] D. J. Tian, A. Bates, and K. Butler. Defending Against Malicious USB Firmware with GoodUSB. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 261–270. ACM, 2015.

[53] D. J. Tian, N. Scaife, A. Bates, K. Butler, and P. Traynor. Making USB Great Again with USBFILTER. In *USENIX Security Symposium*, 2016.

[54] T. Vidas, D. Votipka, and N. Christin. All Your Droid Are Belong to Us: A Survey of Current Android Attacks. In *WOOT*, pages 81–90, 2011.

[55] Z. Wang and A. Stavrou. Exploiting Smart-Phone USB Connectivity For Fun And Profit. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 357–366. ACM, 2010.

[56] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The Impact of Vendor Customizations on Android Security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 623–634. ACM, 2013.

[57] XDA Developers. New Samsung Galaxy S8, S8+, and Note8 Bootloader Prevents Flashing Out of Region Firmware. https://www.xda-developers.com/samsung-galaxy-s8-note8-bootloader-odin/, 2018.

[58] XDA Forums. How to talk to the Modem with AT commands. https://forum.xda-developers.com/galaxy-s2/help/how-to-talk-to-modem-commands-t1471241, 2012.

[59] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 409–423. IEEE, 2014.

A Additional Implementation Details

A.1 AT Extraction Details

Some limitations of our extraction include potentially missing AT commands, images that fail to extract, and missing build.prop files. Given our AT command regular expression and the symbol set we use, we may miss commands using a non-standard symbol following the `AT`. In practice, we observe from AT command standards, existing online AT databases, manual analysis in IDA Pro, Google searches, and more permissive regular expressions that the vast majority of extended AT commands found in the wild are uppercase and use one of the symbols [+*!@#$%^&] matched by our expression. Despite this, if new valid patterns are found in the future, they can be easily added to our regular expression.

Images that fail to extract completely are still analyzed for strings, but if they are compressed, detecting any matches will be impossible. If an image is missing a `build.prop` file, we *do not* include it in our dataset, as this may be indicative of an invalid Android image, since all AOSP images are mandated to contain this file.

A.2 AT Database Filtering

Filtering Heuristic

$$cmd := \texttt{String}$$
$$file := \texttt{AtFile}$$
$$charclass = \begin{cases} e^{-0.4*(cmd::len-3)}, cmd::class \ni \{alnum\} \\ 0, \text{otherwise} \end{cases}$$
$$file\_score = \frac{file::badlines}{file::lines} \cdot \underset{[0,1]}{\text{map}}(e^{0.05*file::lines} - 1)$$
$$at\_score = 10 \cdot (\underset{[0,1]}{\text{map}}(charclass) + file\_score)$$

We define `String` and `AtFile` as types, $var::attr$ as accessing the attribute $attr$ of var, and the

$$\underset{[x,y]}{\text{map}}(n) \text{ function}$$

to clamp n to the range $x \leq n \leq y$.

In practice we observe that it is less common for an AT command to have digits ([0-9]) and lower case letters ([a-z]) in the same command. We punish commands matching this with an exponential decay term in terms of a constant and the command length with the *charclass* metric. The minimum command that would be scored is three (3) characters, hence the subtraction of three. The larger the candidate AT command, the less it is punished, as the likelihood that the command is not random noise increases with each character.

For the $file\_score$ metric, we record every line found that fails the initial regular expression test and increase the $file::badlines$ variable. For each line, regardless of it failing or passing, we increase the $file::lines$ variable. This creates a false positive percentage for the file. We increase the confidence of this FP score exponentially based off of the number of lines seen in the file and a constant of our choosing.

Finally we sum and weight the *charclass* and $file\_score$ metrics to create a final *at_score* (a lower score means that it is less likely to be spurious). For future processing, we set the spurious command threshold to be $at\_score \geq 5.0$. Through manual inspection we found this balances the number of false negatives (actual commands discarded) and false positives (bad commands accepted).

Filtering Results During the initial extraction of firmware images, we used `strings` to match on lines matching the regular expression `AT[!@#$%^&*+]`. To narrow down on actual AT commands, we applied the heavier regular expression, which eliminated 33.2% of all processed lines, as shown in Table 15. To further refine our matching and eliminate classes of frequently-appearing commands, we applied our heuristic to discard additional matches that passed the regular expression. This heuristic eliminated an additional 2.4% of all processed lines and brought the total unique AT command count down from 4,654 to 3,500, a 24.8% reduction. With less invalid commands matched, our analytics were not skewed and our AT command testing was faster.

Due to how the heuristic is implemented, it only has memory of firmware image file score across a single vendor. Also, it is possible for invalid commands to avoid this check by appearing early in a file without a score or in a file with a good score. This is a limitation and could be improved by additional feature checking, multiple passes, and blacklisting. In our work we found the heuristic developed to be sufficient for our purposes. Additionally, we spot-checked the spurious commands and their corresponding *at_score* to make sure that large amounts of valid commands were not being discarded. Overall, we purposefully avoided any manual filtering to make importing new datasets fast and less labor intensive.

A.3 Android Firmware Acquisition

Vendors such as Google and ASUS list all of their factory images for download on their official websites. A combination of URL extraction from the HTML page plus `wget` allows us to efficiently gather and download each image. Other vendors do not provide their Android firmware downloading directly. In these cases, we refer to third-party websites (e.g., AndroidMTK.com) which collect Android firmware images from various vendors.

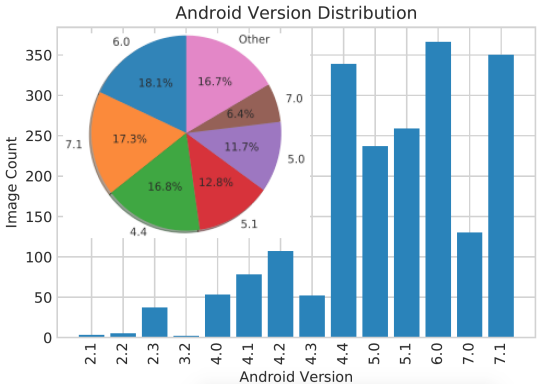


Figure 7: Android Version Distribution.

For these third-parties, the actual download URL is usually found after jumping through multiple site redirections, clicking JavaScript buttons, avoiding rogue download buttons, and passing Turing tests. Images themselves are usually hosted by cloud storage services such as MediaFire or AndroidFileHost. Effectively, all of the images we have gathered are publicly available with some effort on categorizing and collecting valid URLs for download.

Factory/stock firmware is available on the official vendor sites for ASUS, Google, and HTC. For all other vendors, we rely on third-party sites that collect firmware images, among which we choose sites that claim to host only stock firmware. We download firmware from sources listed in Table 12.

Firmware Version Distribution. The Android version distribution across all collected factory images is presented in Figure 7. Versions 4.x, 5.x, 6.x, and 7.x make up the largest percentage of all images, with over 200 images of Versions 4.4, 6.0, and 7.1. We do not prioritize specific versions during the image crawling process. The version distribution of our dataset appears to *reflect mainstream Android devices* that are still in use, e.g., Google Nexus series (4.x and 5.x), LG G series (6.x), and the latest Samsung Galaxy series (7.x). Note that Android 8.x (Oreo) is intentionally excluded since most vendors had not started rolling out their updates by the time of writing.

Vendor	Download URL
ASUS	`https://www.asus.com/support`
Google	`https://developers.google.com/android/images`
HTC	`http://www.htc.com/us/support/rom-downloads.html` `http://www.htc.com/us/support/updates.aspx`
Huawei	`https://androidmtk.com/download-huawei-stock-rom-for-all-models`
Lenovo	`https://androidmtk.com/download-lenovo-stock-rom-models`
LG	`http://devtester.ro/projects/lg-firmwares/`
LineageOS	`https://download.lineageos.org/`
Motorola	`https://firmware.center/firmware/Motorola/`
Samsung	`https://androidmtk.com/download-samsung-stock-rom`
Sony	`http://www.firmwaremobile.com/index.php/xperiadownload/`
ZTE	`https://freeandroidroot.com/download-zte-stock-rom-firmware/`

Table 12: A list of online resources from which we downloaded Android stock firmware.

Aggregation (2018)	Google (447)	Samsung (373)	LG (150)
AT+CLCC (2011)	AT+CGEREP (447)	AT+COPS (373)	AT+WNAM (150)
AT+CHLD (2011)	AT+CSQ (447)	AT+CLCC (373)	AT%GYRO (150)
AT+VTS (2010)	AT+CGDCONT (447)	AT+CGSN (373)	AT%FUSG (150)
AT+COPS (2007)	AT+CHLD (447)	AT+CCWA (373)	AT%NCM (150)
AT+CCWA (2007)	AT+COPS (447)	AT+CHLD (373)	AT%LGATSERVICE (150)
AT+CMEE (2005)	AT+CGREG (447)	AT+VTS (373)	AT%SIMID (150)
AT+CGSN (1996)	AT+CGACT (447)	AT+CMEE (373)	AT%MLT (150)
AT+CMGS (1969)	AT+CMUT (447)	AT+DEVCONINFO (370)	AT+BTRH (150)
AT+CFUN (1968)	AT+CGSN (447)	AT+PROF (368)	AT%MMCFORMAT (150)
AT+CMGW (1967)	AT+CSMS (447)	AT+SYNCML (367)	AT%MDMLOG (150)

Table 13: Top 10 ATcmds (frequency#) in Aggregation, Google, Samsung, and LG.

Command	Action	Tested Phones
ATI	Manufacturer, model, revision, SVN, IMEI	G4/S8+/Nexus5 ZenPhone2
AT%IMEI	IMEI information	G3/G4
AT%SYSCAT	Read and return data from `/sys/*`	G3/G4
AT%PROCCAT	Read and return data from `/proc/*`	G3/G4
AT+BATGETLEVEL?	Battery information	Note2/S7Edge/S8+
AT+CGMM	Phone model	G3/Note2/S8+/ Nexus5/ZenPhone2
AT+CGSN	Serial number	Note2/ZenPhone2/ ZenPad
AT+DEVCONINFO	Phone model, serial number, IMEI, and etc.	Note2/S7Edge/S8+
AT+GMR	Phone model	G3/G4/Note2 S8+/ZenPhone2
AT+GSN	Serial number	G4/Note2/S7Edge/S8+/ ZenPhone2/ZenPad
AT+GSNR	Serial number	Note2/S7Edge/S8+
AT+GSNW	Serial number	Note2/S7Edge/S8+
AT+IMEINUM	IMEI number	Note2/S7Edge/S8+
AT+SERIALNO	Serial number	Note2/S7Edge/S8+
AT+SIZECHECK	Filesystem partition information	Note2/S7Edge/S8+
AT+SVCIFPGM	Partition information and etc.	Note2/S7Edge/S8+
AT+SWVER	Software version	Note2/S7Edge/S8+
AT+GMM	Phone model	G3/G4/S7Edge/S8+/ ZenPhone2
AT+CGMI	Manufacturer identification	G3/S7Edge/S8+/ Nexus5/ZenPhone2
AT+CGMR	Revision identification	G3/S7Edge/S8+/ ZenPhone2
AT+GMI	Manufacturer identification	G3/G4/S8+/ZenPhone2
AT+VERSNAME	Android version	S7Edge/S8+
AT^GSN	Serial number	G4/S8+/Nexus 5
AT^HWVER	Hardware version	G3/G4/S8+/Nexus5
AT^MEID	Serial number	S8+/Nexus5
AT^SYSINFO	System information	S8+/Nexus5
AT+CLAC	List all supported AT commands	G3/G4/S7Edge/Nexus5/ ZenPad/ZenPhone2
AT+LIST	List supported AT commands	Nexus5
AT+ICCID	Sim card ICCID	G3/G4/Nexus5
AT$QCCLAC	List all supported AT commands (Qualcomm-specific)	S8+/G4/Nexus5
AT%SWOV	Software version	G3
AT%SWV	Software version	G3
AT+CGSVN	IMEI information	ZenPhone2
AT+XGENDATA	Software version	ZenPhone2

Table 14: A selection of commands that leak sensitive information about the device.

Vendor	Lines Processed	Matched	Invalid	Spurious
ZTE	25,105	76.3%	21.4%	2.3%
HTC	25,690	24.1%	72.5%	3.3%
Sony	34,390	45.8%	50.2%	4.0%
LineageOS	41,739	62.9%	36.0%	1.2%
Motorola	70,356	50.0%	44.8%	5.3%
Huawei	78,432	79.7%	16.5%	3.8%
Google	133,003	42.1%	51.9%	6.0%
LG	171,578	41.1%	57.1%	1.9%
ASUS	201,996	62.4%	35.2%	2.4%
Lenovo	204,310	81.6%	16.8%	1.6%
Samsung	406,272	76.9%	21.9%	1.2%
Total	1,392,871	64.4%	33.2%	2.4%

Table 15: The results of filtering the lines retrieved by `grep` (Lines Processed) using the AT regular expression in Figure 4 (Matched vs. Invalid) and through applying the *at_score* heuristic (Spurious).