Opening Pandora's Box through ATFuzzer: Dynamic Analysis of AT Interface for Android Smartphones

Imtiaz Karim
Purdue University
karim7@purdue.ede

Fabrizio Cicala
Purdue University
ficiala@purdue.edu

Syed Rafiul Hussain
Purdue University
hussain1@purdue.edu

Omar Chowdhury
University of Iowa
omar-chowdhury@uiowa.edu

Elisa Bertino
Purdue University
bertino@purdue.edu

ABSTRACT

This paper focuses on checking the correctness and robustness of the AT command interface exposed by the cellular baseband processor through Bluetooth and USB. A device's application processor uses this interface for issuing high-level commands (or, AT commands) to the baseband processor for performing cellular network operations (e.g., placing a phone call). Vulnerabilities in this interface can be leveraged by malicious Bluetooth peripherals to launch pernicious attacks including DoS and privacy attacks. To identify such vulnerabilities, we propose ATFuzzer that uses a grammar-guided evolutionary fuzzing approach which mutates production rules of the AT command grammar instead of concrete AT commands. Empirical evaluation with ATFuzzer on 10 Android smartphones from 6 vendors revealed 4 invalid AT command grammars over Bluetooth and 13 over USB with implications ranging from DoS, downgrade of cellular protocol version (e.g., from 4G to 3G/2G) to severe privacy leaks. The vulnerabilities along with the invalid AT command grammars were responsibly disclosed to affected vendors and two of the reported vulnerabilities have been already assigned CVEs (CVE-2019-16400 and CVE-2019-16401).

CCS CONCEPTS

• **Security and privacy → Mobile and wireless security**; *Distributed systems security*; Denial-of-service attacks.

KEYWORDS

Android Smartphone Security and Privacy, Vulnerabilities, Attack

ACM Reference Format:
Imtiaz Karim, Fabrizio Cicala, Syed Rafiul Hussain, Omar Chowdhury, and Elisa Bertino. 2019. Opening Pandora's Box through ATFuzzer: Dynamic Analysis of AT Interface for Android Smartphones. In *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3359789.3359833

1 INTRODUCTION

Modern smartphones operate with two interconnected processing units—an application processor for general user applications and a baseband processor (also known as a cellular modem) for cellular connectivity. The application processor can issue ATtention commands (or, AT commands) [22] through the radio interface layer (RIL, also called AT interface) to interact with the baseband processor for performing different cellular network operations (e.g., placing a phone call). Most of the modern smartphones also accept AT commands once connected via Bluetooth or USB.

Problem and scope. Since the AT interface is an entry point for accessing the baseband processor, any incorrect behavior in processing AT commands may cause unauthorized access to private information, inconsistent system states, and crashes of the RIL daemon and the telephony stack. This paper thus focuses on the following research question: *Is it possible to develop a systematic approach for analyzing the correctness and robustness of the baseband-related AT command execution process to uncover practically-realizable vulnerabilities?*

Incorrect execution of AT commands may manifest in one of the following forms: (1) *Syntactic errors*: the device accepts and processes syntactically invalid AT commands; and (2) *Semantic violations*: the device processes syntactically correct AT commands, but does not conform to the prescribed behavior. Successful exploitations of such invalid commands may enable malicious peripheral devices (e.g., a headset), connected to the smartphone over Bluetooth, to access phones's sensitive information, such as, IMSI (International Mobile Subscriber Identity, unique to a subscriber) and IMEI (International Mobile Equipment Identity, unique to a device), or to downgrade the cellular protocol version or stop the cellular Internet, even when the peripheral is only allowed to access phone's call and media audio.

Prior efforts. Existing approaches [43, 44, 53] strive to identify the types of valid AT commands (i.e., commands with valid inputs/arguments conforming to 3GPP reference [1, 9, 11, 12, 52] or vendor-specific commands [13, 14, 18, 21] added for vendor customization) exposed through USB interfaces on modern smartphone platforms and the functionality they enable. Yet these studies have at least one of the following limitations: (A) The analyses [53] do not test the robustness of the AT interface in the face of invalid commands; (B) The analyses [53] only consider USB interface and thus leave the Bluetooth interface to the perils of both valid and invalid AT commands; and (C) The analyses [30, 43, 45, 49] are not general enough to be applicable to smartphones from different vendors.

Challenges. Conceptually, one can approach our problem using one of the following two techniques: (1) static analysis; (2) dynamic analysis. As the source code of firmwares is not always available, a static analysis-based approach would have to operate on a binary level. The firmware binaries, when available, are often obfuscated and encrypted. Making such binaries amenable to static analysis requires substantial manual reverse engineering effort. To make matters worse, such manual efforts are often firmware-version specific and may not apply to other firmwares, even when they are from the same vendor. Using dynamic analysis-based approaches also often requires instrumenting the binary to obtain coverage information for guiding the search. Like static analysis, such instrumentation requires reverse engineering effort which again is not scalable. Also, during dynamic analysis, due to the separation of the two processors, it is often difficult to programmatically detect observable RIL crashes from the application processor. Finally, in many cases, undesired AT commands are blacklisted [2–4] and hence can induce rate-limiting by completely shutting down the AT interface. The only way to recover from such a situation is to reboot the test device which can substantially slow down the analysis.

Our approach. In this paper, we propose ATFuzzer which can test the correctness and robustness of the AT interface. One of the key objectives driving the design of ATFuzzer is discovering problematic input formats instead of just some misbehaving concrete AT commands. Towards this goal, ATFuzzer employs a grammar-guided evolutionary fuzzing-based approach. Unlike typical mutation-based [28, 35–37, 46, 59] and generation-based [20, 31, 54] fuzzers, ATFuzzer follows a different strategy. It mutates the production rules of the AT command grammars and uses sampled instances of the generated grammar to fuzz the test programs.

Such an approach has the following two clear benefits. First, a production rule (resp., grammar) describing a valid AT command can be viewed as a symbolic representation for a set of concrete AT commands. Such a symbolic representation enables ATFuzzer to efficiently navigate the input search space by generating a diverse set of concrete AT command instances for testing. The diversity of fuzzed input instances is likely achieved because mutating a grammar can move the fuzzer to test a different syntactic class of inputs with high probability. Second, if ATFuzzer can find a problematic production rule whose sampled instances can regularly trigger an incorrect behavior, the production rule can then be used as an evidence which can contribute towards the identification of the underlying flaw that causes the misbehavior.

ATFuzzer takes grammars of AT commands as the seed. It then generates the initial population of grammars by mutating the seed grammars. For each generated grammar, ATFuzzer samples grammar-compliant random inputs and evaluate the fitness of each grammar based on our proposed fitness function values of the samples. Since code-coverage or subtle memory corruptions are not suitable to be used as the fitness function for such vendor-specific, closed-source firmwares, we leverage the execution timing information of each AT command as a loose-indicator of code-coverage information. Based on the fitness score of each grammar, ATFuzzer selects the parent grammars for crossover operation. We design a grammar-aware *two-point* crossover operation to generate a diverse set of valid and invalid grammars. After the crossover operations, we incorporate three proposed mutation strategies to include randomness within the grammar itself. The intuition behind using both crossover and mutation operations is for testing the integrity of each command field as well as the command sequence[1].

Findings. To evaluate the generality and effectiveness of our approach, we evaluated ATFuzzer on 10 Android smartphones (from 6 different vendors) with both Bluetooth and USB interfaces. ATFuzzer has been able to uncover a total of 4 erroneous AT grammars over Bluetooth and another 13 AT grammars over USB. Impacts of these errors range from complete disruption of cellular network connectivity to retrieval of sensitive personal information. We show practical attacks through Bluetooth that can downgrade or shutdown Internet connectivity, and also enable illegitimate exposure of IMSI and IMEI when such impacts are not achievable through valid AT commands. In addition, the syntactically and semantically flawed AT commands over USB can cause crashes, compound actions, and syntactically incorrect commands to get processed. For instance, an invalid AT command `ATDI` in LG Nexus 5 induces the program to execute two valid AT commands— `ATD` (dial) and `ATI` (display IMEI), simultaneously. These anomalies add a new dimension to the attack surface when blacklisting or access control mechanisms are put in place to protect the devices from valid yet menacing AT commands. The vulnerabilities along with the invalid AT command grammars were responsibly disclosed to the affected vendors. Among the discovered vulnerabilities, two of them (i.e., DoS and privacy leaks attacks) have been assigned CVEs (CVE-2019-16400 [5] and CVE-2019-16401 [6]).

Contributions. The paper has the following contributions:

(1) We propose ATFuzzer— an automated and systematic framework that leverages grammar-guided, evolutionary fuzzing for dynamically testing the AT command interface in modern Android smartphones. We have made our framework open-source alongside the corpus of AT command grammars we tested. The tool and its detailed documentation are publicly available at: https://github.com/Imtiazkarimik23/ATFuzzer
(2) We show the effectiveness of our approach by uncovering 4 problematic AT grammars through Bluetooth and 13 problematic grammars through USB interface on 10 smartphones from 6 different vendors.
(3) We demonstrate that all the anomalous behavior of the AT program exposed through Bluetooth are exploitable in practice by adversaries whereas the anomalous behavior of AT programs exposed through USB would be effectively exploitable even when valid but unsafe AT commands are blacklisted. The impact of these vulnerabilities ranges from private information exposure to persistent denial-of-service attacks.

[1]Achieving some cellular network operations through the AT interface (e.g., sending an SMS) may require issuing a sequence of AT commands instead of a single AT command.

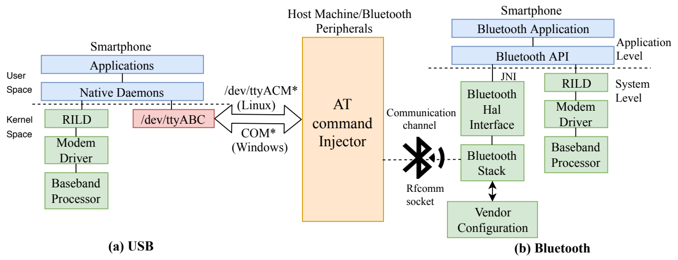

Figure 1: AT Interface for Android Smartphones connected to a host machine through USB interface

2 BACKGROUND

We now give a brief primer on AT commands. We then discuss how we obtain the list of a smartphone-supported AT commands and their respective grammars.

2.1 AT Commands

Along with the AT commands defined by the cellular standards [11], vendors of cellular baseband processors and operating systems support vendor-specific AT commands [17, 18, 21] for testing and debugging purposes. Based on the functionality, different AT commands have different formats; differing in number and types of parameters. The following are the four primary uses of AT commands: (i) Get/read a parameter value, e.g., AT + CFUN? returns the current parameter setting of **+CFUN** which controls cellular functionalities; (ii) Set/write a parameter, e.g., AT + CFUN = 0 turns off (on) cellular connectivity (airplane mode); (iii) Execute an action, e.g., ATH causes the device to hang up the current phone call; (iv) Test for allowed parameters, e.g., AT + CFUN =? returns the allowed parameters for **+CFUN** command. Note that, **+CFUN** is a variable which can be instantiated with different functionality (e.g., **+CFUN**=1 refers to setting up the phone with full functionality).

2.2 AT Interfaces for Smartphones

AT commands can be invoked by an application running on the smartphone, or from another host machine or peripheral device connected through the smartphone's USB or Bluetooth interface (shown in Figure 1). While older generations of Android smartphones used to allow running AT commands from an installed application, recent Android devices have restricted this feature to prevent arbitrary applications from accessing device's sensitive resources illegitimately through AT commands. Contrary to installed applications, nearly all Android phones allow executing AT commands over Bluetooth, whereas, for USB, devices require minimal configuration to set up to activate this feature. Android smartphones typically have different parsers for executing AT commands over these interfaces. The use of different parsers motivates the testing of AT interface for both USB and Bluetooth.

2.3 Issuing AT Commands Over Bluetooth and USB

In this section, we present the details pertaining to issuing AT commands over Bluetooth and USB.

2.3.1 *Bluetooth*.

For executing AT commands over Bluetooth, the injecting host machine/peripheral device needs to be paired with the Android smartphone. The Bluetooth on a smartphone may have multiple profiles (services), but only certain profiles (e.g., hands-free profile, headset profile) support AT commands. Figure 1(b) shows the flow of AT command execution over Bluetooth.

When a device is paired with the host machine, it establishes and authorizes a channel for data communication. After receiving an AT command, the system-level component of the Bluetooth stack recognizes the AT command with the prefix "AT" and compares it against a list of permitted commands (based on the connected Bluetooth profile). When the parsing is completed, the AT command is sent to the application-level component of the Bluetooth stack in user space where the Bluetooth API takes the action as per the AT command issued. Similar to the example through USB, if a baseband related command is invoked (e.g., **ATD <phone_no>;**), the RILD is triggered to deliver the command to the baseband processor. Contrary to USB, only a subset of AT commands related to specific profiles are accepted/processed through Bluetooth.

2.3.2 *USB*.

If a smartphone exposes its USB Abstract Control Model (ACM) interface [49], it creates a tty device such as /dev/ttyACM0 which enables the phone to receive AT commands over the USB interface. On the other hand, in phones for which the USB modem interface is not included in the default USB configuration, switching to alternative USB configuration [49] enables communication to the modem over USB. The modem interface appears as `/dev/ttyACM*` device node in Linux whereas it appears as a `COM*` port in Windows. Figure 1(a) shows the execution path of an AT command over USB.

When the *AT command injector* running on a host machine sends a command through `/dev/ttyACM*` or `COM*` to a smartphone, the `ttyABC` (ABC is a placeholder for actual name of the tty device) device in the smartphone receives the AT command and relays it to the native daemon in the Android userspace. The native daemon takes actions based on the type of command. If the command is related to baseband, for instance, **ATD <phone_no>;**, the RILD (Radio Interface Layer Daemon) is triggered to deliver the command to the baseband processor which executes the command — makes a phone call to the number specified by <phone_no>. On the other hand, if the command is operating system-specific (e.g., Android, iOS, or Windows), such as +CKPD for tapping a key, the native daemon does not invoke RILD.

2.4 AT Commands and Their Grammars

We obtain the list of valid AT commands and their grammars from the 3GPP standards [1, 8–12, 52]. Note that, not every standard AT commands are processed/recognized by all smartphones. This is because different smartphone vendors enforce different whitelisting and blacklisting policies for minimizing potential security risks. Also, vendors often implement several undocumented AT commands. Any problematic input instances that ATFuzzer finds, we check to see whether it is one of the vendor-specific, undocumented AT commands following the approach by Tian et al. [53]. We do not report the undocumented, vendor-specific AT commands that ATFuzzer discovers as invalid since they have already been documented [53]. We aim at finding malformed AT command sets that are due to the parsing errors in the AT parser itself.

3 OVERVIEW OF OUR APPROACH

In this section, we first present the threat model and then formally define our problem statement. Finally, we provide a high-level overview of our proposed mechanism with a running example.

3.1 Threat Model

For Bluetooth and USB AT interfaces exposed by modern smartphones, we define the following two different threat models.

3.1.1 Threat model for Bluetooth. For the threat model over Bluetooth, we assume a malicious/compromised Bluetooth peripheral device (e.g. headphones, speaker) is paired to an Android device. We assume the malicious Bluetooth device is connected through its default profile. For instance, the victim smartphone which is connected to the malicious headphone has only given audio permissions to the headphone. Also, it can be the case when the adversary sets up a fake peripheral device through the man-in-the-middle (MitM) attacks exploiting known vulnerabilities of Bluetooth pairing and bonding [7, 39, 51] procedures. We, however, do not assume the presence of malicious apps on the device.

3.1.2 Threat model for USB. For USB, we assume a malicious USB host, such as a PC or USB charging station controlled by an adversary, that tries to attack the connected Android phone via USB. We assume the attacker can get access to the exposed AT interface even if the device is inactive. We also neither require a malicious app to be installed on the device nor the device's USB debugging to be turned on.

3.2 Problem Statement

Let $\mathcal{I}$ be the set of finite strings over printable ASCII characters, $\mathcal{R} = \{\mathsf{ok}, \mathsf{error}\}$ be the set of parsing statuses, and $\mathcal{A}$ be a set of actions (e.g., phone-call $\in \mathcal{A}$). The AT interface of a smartphone can be viewed as a function $\mathcal{P}$ from $\mathcal{I}$ to $\mathrm{R} \times 2^{(\mathcal{A} \cup \{\mathsf{nop}, \bot\})}$, that is, $\mathcal{P} : \mathcal{I} \to \mathcal{R} \times 2^{(\mathcal{A} \cup \{\mathsf{nop}, \bot\})}$ in which "nop" refers to *no operation* whereas $\bot$ captures undefined behavior including a crash. nop is used to capture the behavior of $\mathcal{P}$ ignoring an AT command, possibly, due to blacklisting or parsing errors.

Given the smartphone AT interface under test $\mathcal{P}_{\mathsf{Test}}$ and a reference AT interface induced by the standard $\mathcal{P}_{\mathsf{Ref}}$, we aim to identify concrete *vulnerable* AT command instances $s \in \mathcal{I}$ such that $\mathcal{P}_{\mathsf{Test}}$ and $\mathcal{P}_{\mathsf{Ref}}$ do not agree on their response for s, that is, $\mathcal{P}_{\mathsf{Ref}}(s) \neq \mathcal{P}_{\mathsf{Test}}(s)$. Given pairs $\langle r_1, a_1\rangle, \langle r_2, a_2\rangle \in \mathcal{R} \times 2^{(\mathcal{A} \cup \{\mathsf{nop}, \bot\})}$, we write $\langle r_1, a_1\rangle = \langle r_2, a_2\rangle$ if and only if $r_1 = r_2$ and $a_1 = a_2$. Note that, a_1 and a_2 are both sets of actions as one command can mistakenly trigger multiple actions.

Note that, there can be a reason $\mathcal{P}_{\mathsf{Ref}}$ and $\mathcal{P}_{\mathsf{Test}}$ can legitimately disagree on a specific input AT command $s \in \mathcal{I}$ as s can be blacklisted by $\mathcal{P}_{\mathsf{Test}}$. Due to CVE-2016-4030 [2], CVE-2016-4031 [3], and CVE-2016-4032 [4], Samsung has locked down the exposed AT interface over USB with a command whitelist for some phones. In this case, we do not consider s to be a vulnerable input instance. Precisely, when s is a blacklisted command, we observed that $\mathcal{P}_{\mathsf{Test}}$ often returns $\langle \mathsf{ok}, \mathsf{nop}\rangle$. Finally, we instantiate the oracle $\mathcal{P}_{\mathsf{Ref}}$ through manual inspection of the standard.

3.3 Running Example

To explain ATFuzzer's approach, we now provide a partial, example context-free grammar (CFG) of a small set of AT commands (see Figure 3 for the grammar and Figure 2 for the partial Abstract Syntax Tree (AST) of the grammar) which we adopted from the original 3GPP suggested grammar [11, 52]. In our presentation, we use the bold-faced font to represent non-terminals and regular-faced font to identify terminals. We use "·" to represent explicit concatenation, especially, to make the separation of terminals and non-terminals in a production rule clear. We use [. . .] to define regular expressions in grammar production rules and [. . .]* to represent the Kleene star operation on a regular expression denoted by [. . .]. In our example, **Dnum** can take any alphanumeric string up to length n as an argument. Our production rules are of the form: $\mathbf{s} \longrightarrow \alpha \cdot \mathbf{B}_1\{\phi\}$ where s is a non-terminal, α denotes a (possibly, empty) sequence of terminals, $\mathbf{B}_1$ represents a possibly empty sequence of non-terminals and terminals, and ϕ represents a condition that imposes additional well-formedness restrictions on the production.

In the above example, we show the correct AT command format for making a phone call. Examples of valid inputs generated from this grammar can be— ATD $*$ 752# + 436644101453;

3.4 Overview of ATFuzzer

In this section, we first touch on the technical challenges that ATFuzzer faces and how we address them. We conclude by providing the high-level operational view of ATFuzzer.

3.4.1 Challenges. For effectively navigating the input search space and finding vulnerable AT commands, ATFuzzer has to address the following four technical challenges.

C1: *(Problematic input representation).* The first challenge is to efficiently encode the pattern of problematic inputs. It is crucial as the problematic AT commands that have similar formats/structures but are not identical may trigger the same behavior. For instance, both ATD123 and ATD1111111111 test inputs are problematic (neither of them is a compliant AT command due to missing a trailing semicolon) and have a similar structure (i.e., ATD followed by a phone number), but are not the same concrete test inputs. While processing these problematic AT commands, one of our test devices, however, stops cellular connectivity. Mutation in the concrete input level will require the fuzzer to try a lot of inputs of the same vulnerable structure before shying away from that abstract input space. This may limit the fuzzer from testing diverse classes of inputs.

C2: *(Syntactic correctness).* As shown in Figure 3, most of the AT commands have a specific number and type of arguments. For instance, **+CFUN**= has two arguments: **CFUNarg1** and **CFUNarg2**. The second challenge is to effectively test this structural behavior and argument types, thoroughly by generating diverse inputs that do not comply with the command structure or the argument types.

C3: *(Semantic correctness).* Each argument of an AT command may have associated conditions. For instance, $Length(\mathbf{Dnum}) \leq n$ in the fifth production rule of Figure 3. Also, arguments may correlate with each other, such as, one argument defines a type on which another argument depends. For instance, **+CTFR**= refers to a service that causes an incoming alert call to be forwarded to a specified number. It takes four arguments— the first two are **number** and **type**, respectively. Interestingly, the second argument defines the

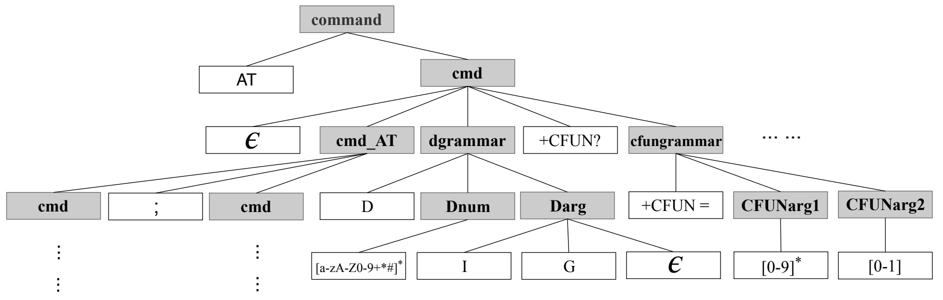

Figure 2: Partial Abstract Syntax Tree(AST) of the reference grammar (Grey-box denotes non-terminal symbols and white box indicates terminal symbols)

$$
\begin{aligned}
\textbf{command} &\rightarrow \mathsf{AT} \cdot \textbf{cmd} \\
\textbf{cmd} &\rightarrow \textbf{dgrammar} \mid \textbf{cfungrammar} \\
\textbf{cmd} &\rightarrow \epsilon \mid \textbf{cmd\_AT} \\
\textbf{cmd\_AT} &\rightarrow \textbf{cmd} \, ; \textbf{cmd} \\
\textbf{dgrammar} &\rightarrow \mathsf{D} \cdot \textbf{Dnum} \cdot \textbf{Darg} \, ; \\
\textbf{cfungrammar} &\rightarrow \mathsf{+CFUN?} \mid \mathsf{+CFUN} = \textbf{CFUNarg1}, \ \textbf{CFUNarg2} \\
\textbf{cmd} &\rightarrow \mathsf{+CTFR} = \textbf{number}, \ \textbf{type}, \ \textbf{subaddr}, \ \textbf{satype} \\
\textbf{Dnum} &\rightarrow [a - zA - Z0 - 9 + *\#]^* \ \ \{Length(\textbf{Dnum}) \leq n\} \\
\textbf{Darg} &\rightarrow \mathsf{I} \mid \mathsf{G} \mid \epsilon \\
\textbf{CFUNarg1} &\rightarrow [0 - 9]^* \ \ \{\textbf{CFUNarg1} \in \mathbb{Z} \text{ and } 0 \leq \textbf{CFUNarg1} \leq 127\} \\
\textbf{CFUNarg2} &\rightarrow [0 - 1] \ \ \{\textbf{CFUNarg2} \in \mathbb{Z} \text{ and } 0 \leq \textbf{CFUNarg2} \leq 1\} \\
\textbf{number}_1 &\rightarrow \textbf{number}_1 \mid \textbf{number}_2 \\
\textbf{number}_2 &\rightarrow [a - zA - Z0 - 9 + *\#]^* \ \{\text{if } \textbf{type} = 145\} \\
\textbf{number} &\rightarrow [a - zA - Z0 - 9 * \#]^* \ \{\text{if } \textbf{type} = 129\} \\
\textbf{type} &\rightarrow 145 | 129 \\
\textbf{subaddr} &\rightarrow [a - zA - Z0 - 9 + *\#]^* \\
\textbf{satype} &\rightarrow [0 - 9]^* \{\text{if satype} = \epsilon, \ \text{satype} = 128\} \\
&\vdots
\end{aligned}
$$

Figure 3: Partial reference context-free grammar for AT commands.

format of the number given as the first argument. If the dialing string includes access code character "+", then the type should be 145, otherwise, it should be 129. These correlations are prevalent in many AT commands. Hence, the third challenge is to systematically test conditions associated with the arguments of commands to cover both syntactical and semantic bugs.

C4: *(Feedback of a test input).* The AT interface can be viewed as a black-box providing only limited output of the form: OK (i.e., correctly parsed) or ERROR (i.e., parsing error). The final challenge is to devise a mechanism that can provide information about the code-coverage of the AT interface for the injected test AT command and thus effectively guiding us through the fuzzing process.

3.4.2 Insights on addressing challenges. For addressing C1, we use the grammar itself as the seed of our evolutionary fuzzing framework rather than using a particular instance (i.e., a concrete test input) of the grammar. This is highly effective as the mutation of a production rule can influence the fuzzer to test a diverse set of inputs. Also, when a problematic grammar is identified, it can serve as abstract evidence of the underlying flaw in the AT interface. Finally, as grammar can be viewed as a symbolic representation of concrete input AT commands, mutating a grammar can enable the fuzzer to cover large diverse classes of AT commands. The insight here is that testing diverse input classes are likely to uncover diverse types of issues.

To address challenges C2 and C3, at each iteration, ATFuzzer chooses parents with the highest fitness scores and switches parts of the grammar production rules among each other. This causes changes to not only the structural and type information in the child grammars but also forms two very different grammars that try to break the correlation of the arguments. For instance, suppose that the ATFuzzer has selected following two production rules from two different parent grammars: **+CFUN** = **CFUNarg1**, **CFUNarg2** and **+CTFR** = **number**, **type**, **subaddr**, **satype**. After applying our proposed grammar crossover mechanisms, the resultant child grammar production rules are: **+CFUN** = **CFUNarg1**, **type**, **subaddr**, **satype** and **+CTFR** = **number** **CFUNarg2**. The production rule **+CFUN** takes only two arguments whereas our new child grammar creates a production rule that has four arguments. The same reasoning also applies for **+CTFR**. Thus, the new grammars with modified production rules would test this structural behavior precisely. Furthermore, **+CTFR**'s first argument **number** is correlated with its second argument **type**. In the modified child grammars, **type**, however, has been replaced with **CFUNarg2**. Recall from our grammar definition, **type** takes argument from the set $\{145, 129\}$ whereas **+CFUNarg2** takes argument from the set $\{0, 1\}$. Therefore, this single operation completes two tasks at once—it not only tests the correlation among two arguments of **+CTFR** but also tests conditions of both **+CFUN** and **+CTFR**. Crossing over grammar production rules creates a drastic modification in the input format and it aims to explore the diverse portions of the input space to create highly unusual inputs. To test both the structural aspects, we use three very different mutation strategies which create little change to the grammar (compared to crossover) but prove highly effective for checking the robustness of the AT interface.

For addressing **C4**, we use the precise timing information of injecting an AT command and receiving its output. We keep an upper bound on this time, i.e., a timer ($\mathcal{T}$). If the output is not received within $\mathcal{T}$, we suspect the AT interface has become unresponsive possibly due to the blacklisting mechanism enforced by several vendors. We use this timing information as a *loose-indicator* for the code-coverage information. Our intuition is to explore as much of the AT interface as possible. A high execution time loosely indicates that the test command traverses more basic-blocks than the other inputs with lower execution time. We try to leverage this simple positive correlation to design a feedback edge (i.e., a fitness function) of the closed-loop. The timing information, however, cannot help to infer how many new basic-blocks a test input could explore. Since our focus is mainly on baseband related AT commands, an error in the AT interface has a higher probability

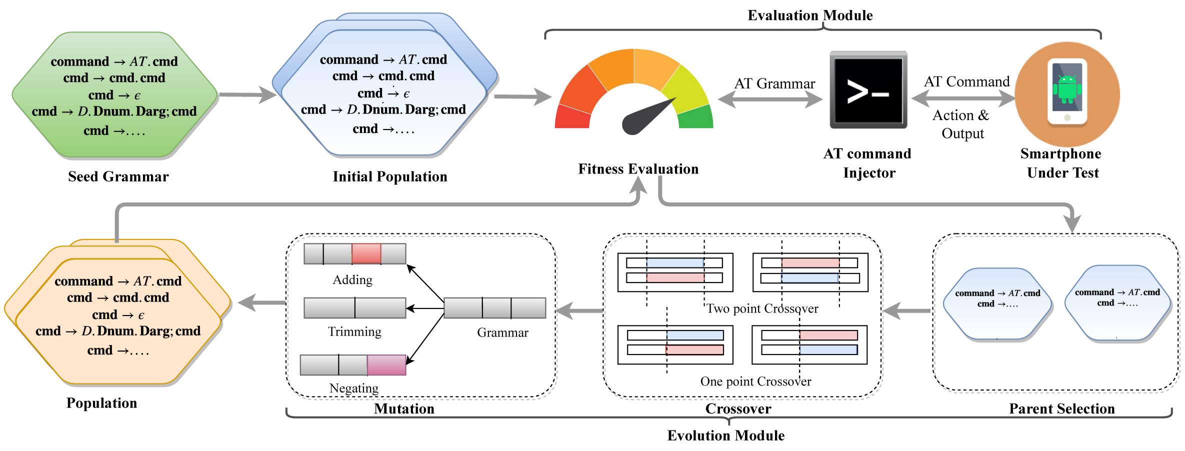

Figure 4: Overview of ATFuzzer **framework**

of causing disruptions in the baseband which also trickles down to cellular connectivity. We leverage this key insight and consider both the cellular Internet connectivity information from the target device and the device's debug information (*Logcat*) as an indication of the baseband health after running an AT command. Using this information, we devise our fitness function for guiding ATFuzzer.

3.4.3 High-level description of ATFuzzer. ATFuzzer comprises of two modules, namely, *evolution module* and *evaluation module*, interacting in a closed-loop (see Figure 4). The evolution module is bootstrapped with a seed AT command grammar which is mutated to generate P_{size} (refers to population size; a parameter to ATFuzzer) different versions of that grammar. Concretely, new grammars are generated from parent grammar(s) by ATFuzzer through the following high-level operations: (1) Population initialization; (2) Parent selection; (3) Grammar crossover; (4) Grammar mutation. Particularly relevant is the operation of *parent selection* in which ATFuzzer uses a *fitness function* to select higher-ranked (parent) grammars for which to apply the crossover/mutation operations (i.e., steps 3 and 4) to generate new grammars. Choosing the higher-ranked grammars to apply mutation is particularly relevant for generating effective grammars in the future.

Evaluating fitness function requires the evaluation module. For a given grammar g, evaluation module samples several g-compliant commands to test. It uses the *AT command injector* (as shown in Figures 1 and 4) to send these test commands to the device-under-test. The fitness function uses the individual scores of the concrete g-compliant instances to assign the overall score to g.

4 DETAILED DESIGN OF ATFUZZER

In this section, we discuss our proposed crossover and mutation techniques for the *evolution module* followed by the fitness function design used by the *evaluation module*.

4.1 Evolution Module

Given the AT grammar (shown in Figure 3), ATFuzzer's evolution module randomly selects at most n **cmd**s to generate the initial seed AT grammar denoted as G_{AT}. The evolution module yields the grammars G_{best} with the highest scores until a certain stopping condition is met, such as, total time of testing or the number of iterations. Algorithm 1 describes the high-level steps of ATFuzzer's *evolution module*.

4.1.1 Initialization. The *evolution module* starts with initializing the population P (Line 1 in Algorithm 1) by applying both our proposed crossover and mutation strategies with three parameters: the population size P_{size}; the probability $\mathcal{P}_{pop}$ of applying crossover and mutation on the grammar; the tournament size T_{size}. The key-insight of using $\mathcal{P}_{pop}$ is that it correlates with the number of syntactic and semantic bugs explored. A higher value of $\mathcal{P}_{pop}$ is, the diverse the initial population is and vice versa. The diverse the initial population, the higher the number of test inputs that check syntactic correctness is and vice versa. Therefore, to explore both syntactic and semantic bugs, we vary the values of $\mathcal{P}_{pop}$; aiming to strike a balance between grammar diversity. To assess the fitness of the initial population P, the evolution module invokes the evaluation module (Line 3-8) with the generated grammars.

Algorithm 1: ATFuzzer

```
   Data: $P_{size}$, $\mathcal{P}_{pop}$, $G_{AT}$, $T_{size}$
   Result: $G_{best}$: Best Grammar
 1 P ← InitializePopulation($P_{size}$, $\mathcal{P}_{pop}$, $G_{AT}$);
 2 while stopping condition is not met do
 3     for each grammar $G_i$ ∈ P do
 4         Generate random input I
 5         AssesFitness($G_i$, I)
 6         if Fitness($G_i$) > Fitness($G_{best}$) then
 7             $G_{best}$ = $G_i$;
 8         end
 9     end
10     Q = {}
11     for $\frac{P_{size}}{2}$ times do
12         $P_a$ ← ParentSelection(P, $T_{size}$)
13         $P_b$ ← ParentSelection(P, $T_{size}$)
14         $C_a$, $C_b$ ← GrammarBasedCrossover($P_a$, $P_b$)
15         Q = Q ∪ {Mutate($C_a$), Mutate($C_b$)}
16     end
17     P ← Q
18 end
```

4.1.2 Parent selection for the next round. We use the *tournament selection* technique to get a diverse population at every round. We perform "tournaments" among P grammars (Line 12-13 in Algorithm 1). The winner of each tournament (the one with the highest fitness score) is then selected for crossover and mutation. In what follows, we discuss in detail our *tournament selection* technique addressing the functional and structural bloating problems of evolutionary fuzzers [54].

Restraining functional bloating. We leverage another insight in selecting grammars at each round of the tournament selection procedure to reduce *functional bloating* [54]— the continuous generation of grammars containing similar mutated production rules—which adversely affects diverse input generation in evolutionary fuzzing. At each round, we *randomly* select grammars from our population. This is due to the fact that while running an evolutionary fuzzing, the range of fitness values becomes narrow and reduces the search space it focuses on. For example, at any round, if the fuzzer finds a grammar that has a mutated production rule related to **+CFUN** causing an error state in the AT interface, then all the grammars containing this mutated rule will obtain high fitness values. If we then only select parents based on the highest fitness, we would inevitably fall into functional bloating and would narrow down our focused search space with grammars that are somehow associated with this mutated version of **+CFUN** only.

To constraint this behavior, we perform the tournament selection procedure in which we randomly choose T_{size} (where T denotes the set of selected grammars for the tournament and $T_{size} \leq P_{size}$) number of grammars from the population P. The key insight of choosing *randomly* is to give chances to the lower fitness grammars in the next round to ensure a diverse pool of candidates with both higher and lower fitness scores.

Restraining structural bloating. After running ATFuzzer for a while, i.e., after a certain number of generations, the average length of individual grammar grows rapidly. This behavior is characterized as *structural bloating*. Referring to the AT grammar in Figure 3, multiple **cmd**s (production rules) can contribute to generating the final commands that are sent to the AT command injector for evaluation. These commands can grow indefinitely, but do not induce any structural changes, and thus cause structural bloating. These input commands, therefore, hardly contribute to the effectiveness of the fuzzer. To limit this behavior, we restrict the grammar to have at most three **cmd**s at each round to generate the input AT commands for testing.

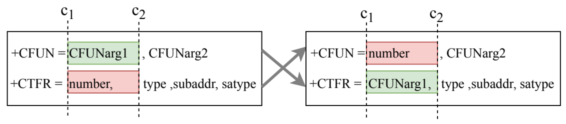
c1
c2
+CFUN = CFUNarg1 , CFUNarg2
+CTFR = number, type ,subaddr, satype
c1
c2
+CFUN = number , CFUNarg2
+CTFR = CFUNarg1, type ,subaddr, satype

Figure 5: Examples of one-point and two-point grammar cossover mechanisms.

4.1.3 Grammar crossover. In the grammar crossover stage, ATFuzzer strives to induce changes in the grammar aiming to systematically break the correlation and structure of the grammar. For this, we take inspiration from traditional genetic programming and apply our custom two-point crossover technique to the grammars.

Two-point crossover. ATFuzzer picks up two random production rules from the given parent grammars and generates two random numbers c_1 and c_2 within ℓ where ℓ is the minimum length between the two production rules. ATFuzzer then swaps the fields of the two production rules that are between points c_1 and c_2.

Figure 5 shows how ATFuzzer performs the two-point crossover operation on production rules +CTFN = and +CTFR = (a subset of the AT grammar in Figure 3) used for controlling the cellular functionalities and for urgent call forwarding, respectively. By applying two-point crossover on +CFUN = **CFUNarg1, CFUNarg2** and +CTFR = **number, type, subaddr, satype**, ATFuzzer generates +CFUN = **number, CFUNarg2** and +CTFR = **CFUNarg1, type, subaaddr, satype** which in turn contribute in generating versatile inputs.

Algorithm 2: Two-Point Grammar Crossover

```
   Data: ParentGrammar $P_a$, ParentGrammar $P_b$
   Result: $P_a$,$P_b$
 1 Randomly pick production rule $R_a$ from $P_a$ and $R_b$ from $P_b$
 2 $R_a \leftarrow R_{a_1}, R_{a_2}, ..., R_{a_l}$
 3 $R_b \leftarrow R_{b_1}, R_{b_2}, ..., R_{b_m}$
 4 $c_1 \leftarrow$ random integer chosen from 1 to $min(l, m)$
 5 $c_2 \leftarrow$ random integer chosen from 1 to $min(l, m)$
 6 if $c > d$ then
 7     swap $c_1$ and $c_2$
 8 end
 9 for $i$ from $c$ to $d - 1$ do
10     swap grammar rules of $R_{a_i}$, $R_{b_i}$
11 end
```

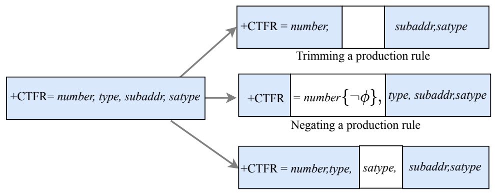
+CTFR = number, subaddr,satype
Trimming a production rule
+CTFR= number, type, subaddr, satype
+CTFR = number{¬ϕ}, type, subaddr,satype
Negating a production rule
+CTFR = number,type, satype, subaddr,satype
Adding a production rule

Figure 6: Example of three grammar mutation strategies.

4.1.4 Grammar mutation. During crossover operation, ATFuzzer constructs grammars that may have diverse structures which are, however, not enough to test the constraints and correlations associated with a command and its arguments. This is due to the fact that AT commands have constraints not only on the fields but also on the commands itself. Therefore, generating versatile grammars that can generate such test inputs is an important aspect of ATFuzzer design. To deal with this pattern, we propose three mutation strategies— *addition*, *trimming*, and *negation*. We use **AT+CTFR=number,type,subaddr,satype** (one of the example grammars presented in Figure 3) to illustrate these mutation strategies with examples shown in Figure 6.

Addition. With our first strategy we randomly insert/add a field chosen from the production rule of the given grammar at a random location. For instance, applying this mutation strategy (shown in Figure 6) to one of the grammars +CTFR = **number,type,subaddr, satype** yields +CTFR = **number,type,satype,subaddr,satype** containing an additional argument added after the second argument of the actual grammar. The mutation also has changed the type (string) of third argument (subaddr) of the actual grammar to integer (satype) in the new grammar. ATFuzzer, thereby, tests the type correctness along with the structure of the grammar.

Trimming. Our second mutation strategy is to randomly trim an argument from a production rule for the given grammar. Referring to Figure 6, applying this to our example grammar for +CTFR, we obtain a production rule AT + CTFR = **number,subaddr,satype** which also deviates from the original grammar with respect to both the structure and type.

Negation. Our last mutation strategy focuses on the constraints associated with the arguments of a command. Referring to the AT grammar in Figure 3, we encode the constraints with additional conditions (denoted with $\{\ldots\}$) in the grammar production rules. With the negation strategy, we randomly pick a production rule of the grammar and choose a random argument that has a condition associated with it. We negate the condition which we use to replace the original one at its original place in the production rule. Figure 6 demonstrates how we negate the production rule associated with number used to represent a phone number. The number is a string type with a constraint on its length. We negate this condition with the following three heuristics: (i) Generating strings that are longer than the specified length; (ii) Generating strings that contain not only alphanumeric characters but also special characters; and (iii) Generating an empty string.

Algorithm 3: Grammar Mutation

```
    Data: Grammar $G_a$, Tunable parameters : $P_\alpha$, $P_\beta$, $P_\gamma$
    Result: Mutated $G_a$
 1  Randomly pick production rule $R_a$ from $G_a$
 2  $R_a \leftarrow R_{a_1}, R_{a_2}, \ldots, R_{a_l}$
 3  $c \leftarrow$ random integer chosen from 1 to $l$
 4  $P \leftarrow$ Generate random probability from (0, 1)
 5  if $P_\alpha \geq P$ then
 6      trim argument $R_{a_c}$ from production rule $R_a$
 7  end
 8  if $P_\beta \geq P$ then
 9      replace $P_c\{\phi\}$ with $P_c\{\neg\phi\}$in production rule $R_a$
10  end
11  if $P_\gamma \geq P$ then
12      $d \leftarrow$ random integer chosen from 1 to $l$
13      add argument $P_{a_c}$ at position $l$ in production rule $R_a$
14  end
```

4.2 Evaluation Module

The primary task of the evaluation module is to generate a number of test inputs (i.e., concrete AT command instances) for the grammar received from the evolution module. It then evaluates the test inputs with the *AT command injector*, and finally evaluate the grammar itself based on the scores of the generated test inputs. To what follows, we explain how the evaluation module calculates the fitness score of a grammar.

4.2.1 Fitness evaluation. At the core of ATFuzzer is the fitness function that guides the fuzzing and acts as a liaison for the coverage information. We devise our fitness function based on the timing information and baseband related information of the smartphone. Our fitness function comprises of two parts: (1) Fitness score of the test inputs generated from a grammar; (2) Fitness score of the grammar in the population.

Fitness score of the test inputs of a grammar. The fitness evaluator of ATFuzzer generates N inputs from each grammar and calculates the score for each input. We define this fitness function for an input AT command instance x as:

$$\boxed{\text{fitness}(x) = \alpha \times \text{timing}_{\text{score}} + (1-\alpha) \times \text{disruption}_{\text{score}}}$$

where α is a tunable-parameter that controls the impact of $\text{timing}_{\text{score}}$ and $\text{disruption}_{\text{score}}$. Let t_x be the time required for executing an AT command x $(0 \leq x < N)$ on the smartphone under test. *Execution time* of an AT command is defined as the time between when the AT command is sent and when the output is received by the AT command injector. Note that, we normalize the execution time by the input length.

Let $t_1, t_2, \ldots, t_N$ be the time for executing N AT commands, we define the timing score for instance x in a population of size N as follows: $\boxed{\text{timing}_{\text{score}} = \frac{t_i}{t_1 + t_2 + \ldots. + t_N}}$.

Note that while running AT commands over Bluetooth, the commands and their responses are transmitted in over-the-air (OTA) Bluetooth packets. To compute the precise execution time of the AT command on the smartphone, we take off the transmission and reception times from the total running time. Also, to make sure Bluetooth signal strength change does not interfere with the timing information, our system keeps track of the RSSI (Received Signal Strength Indication) value and carries out the fuzzing at a constant RSSI value.

We define $\text{disruption}_{\text{score}}$ based on the following four types of disruption events: (i) Complete shutdown of SIM card connectivity; (ii) Complete shutdown of cellular Internet connectivity; (iii) Partial disruption in cellular Internet connectivity; (iv) Partial disruption of SIM card connectivity with the phone. For cases (i) and (ii), complete shutdown causes denial of cellular/SIM functionality, recovery from which requires rebooting the device. ATFuzzer uses **adb reboot** command which takes $\sim 15-20$ seconds to restart the device without entailing any manual intervention. On the contrary, partial shutdown for the cases (iii) and (iv) induce denial of cellular/SIM functionality for $\sim 3-5$ seconds and thus does not call for rebooting the device to recuperate back to its normal state. These events are detected and monitored using the open-source tools available to us from Android, e.g., *logcat*, *dumpsys*, and *tombstone*. When injecting the AT commands we use these tools to detect the events at run time. We take into account if there is a crash in the baseband or the RIL daemon. We assign a score between $0-1$ to a disruption event in which 0 denotes *no disruption at all* (i.e., the device is completely functional) with no adverse effects and 1 denotes complete disruption of the cellular or SIM card connectivities.

Fitness score of a grammar. After computing the fitness scores for all the concrete input instances, we calculate the grammar's score by taking the average over all instance scores.

5 EVALUATION

Our primary goal in this section is to evaluate the effectiveness of ATFuzzer by following the best possible practices [26, 55] and guidelines [34]. We, therefore, first discuss the experiment setup and evaluation criteria, and then evaluate the efficacy of our prototype against the widely used AFL [59] fuzzer— customized for our context.

5.1 Experiment Setup

ATFuzzer setup. We implemented ATFuzzer with ~4000 lines of Python code. We encoded the grammars (with JSON) for a corpus of 90 baseband-related AT commands following the specification in the 3GPP [11] documentation and extracting some of the vendor-specific AT commands following the work of Tian et al. [53]. During its initialization, ATFuzzer receives the name of the AT command as input, retrieves the corresponding grammar that will be used as the seed (G_{AT} in algorithm 1) from the file, generates the initial grammar population, and realizes the proposed crossover and mutation strategies. Hence, our approach is general and easily adaptable to other structured inputs, since it is not bound to any specific grammar structure. Since testing a concrete AT command instance requires 15-20 seconds on average (because of checking the cellular and SIM card connectivity after executing a command and for rebooting the device in case of AT interface's unresponsiveness for blacklisting), we set P_{size} to 10 which we found through empirical study to be the most suitable in terms of ATFuzzer's stopping condition. Following the same procedure, we test 10 concrete AT commands in each round for a given grammar. We set the probability $\mathcal{P}_{pop}$ to 0.5 to ensure uniform distribution in the grammar varying ratio.

Conceptually, one can argue for testing at a "batch" mode to chop the average time for fuzzing an AT grammar. For instance, injecting 10 AT commands together and then checking the cellular and SIM connectivity at once. Although this design philosophy is intuitive, it fails to serve our purpose due to the following fact. This approach may be able to detect permanent disruptions but it is unable to detect temporary disruptions to cellular or SIM connectivity. For instance, even if the second AT command in the batch induces a temporary disruption, there will be no trace of disruptions at all by the time when the tenth (i.e., the last) AT command is executed.

Target devices. We tested 10 different devices (listed in Table 1) from 6 different vendors running 6 different android versions to diversify our analysis. For Bluetooth, we do not require any configuration on the phone. For running AT commands over USB some phones, however, require additional configuration. For additional details, see Appendix A.1.

Device	Android Version	Build Number	Baseband Vendor	Baseband	USB Config	OS	Interface
Samsung Note2	4.3	JSS15J. I9300XU GND5	Samsung Exynos 4412	N7100DD UFND1	None	Linux	Bluetooth and USB
Samsung Galaxy S3	4.3	JSS15J. I9300XX UGND5	Samsung Exynos 4412	I9300XX UGNA8	None	Linux	Bluetooth and USB
LG G3	6.0	MRA58K	Qualcomm Snapdragon 801	MPSS.DI.2.0.1. c1.13-00114 -M8974AA AAANPZM-1.43646.2	None	Linux	Bluetooth and USB
HTC Desire 10 lifestyle	6.0.1	1.00.600.1 8.0_g CL800193 release-keys	Qualcomm Snapdragon 400	3.0.U205591 @60906G_01. 00.U0000. 00_F	sys.usb.config mtp,adb,diag, modem, modem_mdm, diag_mdm	Windows	Bluetooth and USB
LG Nexus 5	5.1.1	LMY48I	Qualcomm Snapdragon 800	M8974A-2.0.50.2.26	sys.usb.config diag,adb	Linux	Bluetooth and USB
Motorola Nexus 6	6.0.1	MOB30M	Qualcomm Snapdragon 805	MDM9625_ 104662.22. 05.34R	fastboot oem bp-tools-on	Windows	Bluetooth and USB
Huawei Nexus 6P	6.0	MDA89D	Qualcomm Snapdragon 810	.2.6.1.c4-00004-M899 4FAAAAN AZM-1	fastboot oem enable-bp-tools	Windows	Bluetooth and USB
Samsung Galaxy S8+	8.0.0	R16NW.G95 5USQU5CRG3	Qualcomm Snapdragon 835	G955US QU5CRG3	None	Linux	Bluetooth and USB
Huawei P8 Lite ALE-L21	5.0.1	ALE-L21CO2B140	HiSilicon Kirin 620 (28 nm)	22.126.12.00.00	None	None	Bluetooth
Pixel 2	8.0.0	OPD3.1708 16.012	Qualcomm MSM8998 Snapdragon 835	g8998-00122-1708231715	None	None	Bluetooth

Table 1: List of the devices we tested, with software information, USB configuration required and the operating system we used to fuzz each device.

5.2 Evaluation Criteria

ATFuzzer has three major components— grammar crossover, mutation, and feedback loop— to effectively test a target device. We, therefore, aim to answer the following research questions to evaluate ATFuzzer:

- **RQ1:** How proficient is ATFuzzer in finding errorneous AT commands over Bluetooth?
- **RQ2:** How proficient is ATFuzzer in finding errorneous AT commands over USB?
- **RQ3:** How effective is our grammar-aware crossover strategy?
- **RQ4:** How effective are our grammar-aware mutation strategies?
- **RQ5:** When using grammars, how much does the use of timing feedback improve fuzzing performance?
- **RQ6:** Is ATFuzzer more efficient than other state-of-the-art fuzzers for testing AT interface?

To tackle **RQ1**-**RQ2**, we let our ATFuzzer run over USB and Bluetooth each for one month to test 10 different smartphones listed in Table 1. ATFuzzer has been able to uncover a total of 4 erroneous AT grammars inducing a crash, downgrade, and information leakage over Bluetooth and 13 erroneous AT grammars over USB. Based on the type of actions and responses to the problematic AT command instances, we initially categorize our results as syntactic and semantic problematic AT grammars, and further categorize the syntactically problematic grammars into three separate classes— (i) responds ok with composite actions; (ii) responds ok with an action; (iii) responds error with an action. Here, an action can either signify a crash (i.e., any disruption event defined in Section 4), leakage of any sensitive information, or executing a command (e.g., hanging up a phone call).

We summarize ATFuzzer's findings for Bluetooth in Table 2 and for USB in Table 3. To answer the research questions **RQ3**-**RQ5**, we evaluate ATFuzzer by disabling one of its components at a time. We create three new instances of ATFuzzer— ATFuzzer without crossover, ATFuzzer without mutation, and ATFuzzer without fitness evaluation. To what follows we evaluate these three variants with the AT grammar (in Figure 3) and compare their efficacy of discovering bugs against original ATFuzzer with all capabilities switched on. Moreover, to answer the research question **RQ6**, we create our variation of AFL (American Fuzzy Lop). To perform a fair comparison, we run all our experiments on Nexus 5 for each variations of ATFuzzer and our version of AFL each for 3 days.

5.3 Findings Over Bluetooth (RQ1)

Unlike USB, Bluetooth does not require any pre-processing or configuration to the phone to execute AT commands. Besides this, over-the-air Bluetooth communications are inherently vulnerable to MitM attacks [7, 39, 51]. All these enable the adversary to readily exploit the vulnerabilities over Bluetooth with sophisticated attacks.

5.3.1 *Results.*

We first discuss the results that relate to invalid AT commands and then we discuss the attacks and impacts of both invalid and valid AT commands.

Class of Bugs	Grammar and Command Instance	action/implication	Nexus5	LG G3	Nexus6	Nexus6P	HTC	S8plus	S3	Note2	Huawei P8lite	Pixel 2
Syntatctic – returns OK with action	**cmd** →D.**Dnum**.**Darg1**.**Darg2** **Dnum** →$[A-Z0-9+\#]^*$ **Darg1** →$I\|G\|\epsilon$ **Darg2** →;. **Darg3** **Darg3** →$[A, B, C]^+$ E.g. ATD + 4642048034I; AB; C	crash/internet connectivity disruption	✓									✓
	cmd →D.**Dnum**.**Darg1**.**Darg2** **Dnum** →$[A-Z0-9+\#]^*$ **Darg1** →$I\|G\|\epsilon$ **Darg2** →;. **Darg3** **Darg3** →$[A, B, C]^+$ E.g. ATD + 4642048034I; AB; C	crash/downgrade			✓	✓						
	cmd →+CIMI.**Arg1** **Arg1** →$[a-zA-Z0-9+\#]^*$ E.g. AT + CIMI; ; ; ; abc	read/IMSI leak						✓	✓	✓		
	cmd →+CGSN.**Arg1** **Arg1** →$[a-zA-Z0-9+\#]^*$ E.g. AT + CGSN; ; ; ; abc##	read/IMEI leak						✓	✓	✓		
Correctly formatted command	**cmd** →+CIND?	read/leaks call status, call setup stage, internet service status, signal strength, current roaming status, battery level, call held status	✓	✓	✓	✓		✓	✓	✓	✓	✓
	cmd →+CHUP	execution (cutting phone calls)/ DoS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	cmd →**Arg**.D.**Dnum**.**Darg**; **Arg** →$[a-zA-Z]$ **Dnum** →$[a-zA-Z0-9+*\#]^*$ **Darg** →$I\|G\|\epsilon$ E.g. ATD $**61*$ +1812555673 $*$ 11 $*$ 25#;	execution/ call forwarding, activating do not disturb mode, activating selective call blocking	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 2: Summary of ATFuzzer**'s Bluetooth parser findings.**

(1) Syntactic errors – responds ok **with actions.** ATFuzzer uncovered four problematic grammars in these categories in seven different Android smartphones. We observer that the target device responds to the invalid AT command and also performs an action. For instance, ATFuzzer found a specific variant of ATD grammar ATDA; A; B in Nexus 5 which is syntactically incorrect, but returns OK and make the cellular Internet connectivity temporarily unavailable. Beside this, the concrete instances of the same grammar also downgrade the cellular connectivity from 4G to the 3G/2G in Nexus 6 and Nexus 6P smartphones thus entails severe security and privacy impacts.

5.3.2 *Attacks with invalid AT commands.*

We now present three practical attacks that can be carried out using the invalid grammars uncovered through ATFuzzer.

Denial of service. The adversary using a malicious Bluetooth peripheral device (e.g., Bluetooth headphone with only call audio and media permissions) or a MitM instance may exploit the invalid AT command, e.g., ATDB; A; B and temporarily disrupt the Internet connectivity of the Pixel 2 and Nexus 5 phones. To cause long term disruptions in Internet connectivity, the adversary may inject this command intermittently and thus prevent the user from accessing the Internet. Note that, there is no valid AT command that controls the Internet connectivity over Bluetooth and thus it is not possible to achieve this impact using a valid AT command.

Downgrade. The same invalid grammar (shown in table 2) exploited in the previous DoS attack in Nexus 5 phone can also be exploited to downgrade the cellular connectivity on Nexus 6 and Nexus 6P phones. Similar to the previous DoS attack, such downgrade of cellular connectivity is not possible with any valid AT commands running over Bluetooth. Downgrade (also known as *bidding-down*) attacks have catastrophic implications as they open the avenue to perform over-the-air man-in-the-middle attacks in cellular networks [25, 40].

IMSI & IMEI catching. ATFuzzer uncovered the invalid variations (AT + CIMI; ; ; ; ; abc and AT + CGSN123df) of two valid AT commands (+CIMI and +CGSN) which enable the adversary to illegitimately capture the IMSI and IMEI of a victim device over Bluetooth. Exploiting this, any Bluetooth peripheral connected to the smartphone can surreptitiously steal such important personal information of the device and the network. We have successfully validated this attack in Samsung Galaxy S3, Samsung Note 2, and Samsung Galaxy S8+. One thing to be noted here is that after manual testing we found out that the valid versions of these two commands also leak IMSI and IMEI. We argue that even if there is a blacklist/firewall policy put into place to stop the leakage through valid AT commands, yet it will not be sufficient because it will leave the scope to use the invalid versions of the command (that ATFuzzer uncovered) to expose this sensitive information.

The impact of this attack is particularly more fatal than that of the previous two attacks. This is because the illegitimate exposure of IMSI and IMEI through Bluetooth provides an edge to the adversary to further track the location of the user or intercept phone calls and SMS using fake base stations [32, 33] or MitM relays [50]. Samsung has already acknowledged the vulnerabilities and is working on issuing patches to the affected devices. We also summarize the findings of ATFuzzer in Table 2. CVE-2019-16401 [6] has been assigned to this vulnerability along with other sensitive information leakage for the affected Samsung devices.

5.3.3 *Attacks with valid AT commands.*

We summarize ATFuzzer's other findings in which we demonstrate that the exposed AT interface over Bluetooth allows the adversary to run valid AT commands to attain malicious goals that may negatively affect a device's expected operations. The results are particularly interesting as Bluetooth interface has not yet been *systematically* examined.

Information leak. The adversary can use a valid AT command to learn the whole set of private information about the phone. The malicious Bluetooth peripheral device can get the call status, call setup state, Internet service status, the signal strength of the cellular network, current roaming status, battery level, and call hold status for the phone using this valid AT commands.

DoS attacks. A malicious peripheral can exploit the AT + CHUP command to prevent the victim device from receiving any incoming phone call. From the previous *information leakage* (e.g., call status) attack, an attacker can probe periodically to detect whether there is a phone call or not. Whenever he detects there is a phone call, the attacker injects AT + CHUP to cut the phone call. To make the matters worse, the attack is transparent to the victim, i.e., there is no indication on the mobile screen that an attack is going on. The victim device user perceives either there is no incoming call or abrupt call drops due to poor signal quality or network congestions. CVE-2019-16400 [5] has been assigned for this along with other reported denial of service attacks in Samsung phones.

Call forwarding. If the victim device is subscribed to call forwarding service, the adversary may exploit the ATD command to forward victim device's incoming calls to an attacker-controlled device. Exploiting this, the adversary first prevents the victim device from receiving the incoming calls and then learns sensitive information, such as password or pin for two-factor authentication possibly sent by an automated teller. Note that, such call forwarding is also transparent to the user since the user is unaware of any incoming calls.

Activating *do not disturb mode.* The adversary using a malicious Bluetooth peripheral or MitM instance can turn on the *do not disturb* mode of the carrier through ATD command. Similar to call forwarding attack, it is also completely transparent to the user as no visible indication of *do not disturb* mode is displayed on the device. While the user observes all the network status bars and the Internet connectivity, the device, however, does not receive any call from the network.

Selective call blocking. A variation of the previous attack is also possible in which the adversary may allow the victim phone to receive selective calls by intermittently turning on/off the *do not disturb* mode. This may force the user to receive calls only from selective users not affecting others.

5.4 Findings over USB (RQ2)

We now discuss findings over USB.

(1) Syntactic errors – responds ok **with composite actions.** It is one of the interesting classes of problematic grammars for which the AT interface of the affected devices respond to invalid AT commands with ok, but performs multiple actions together. These invalid commands are compositions of invalid characters and two valid AT commands with no semicolon as their separator. For instance, ATFuzzer generated an invalid command ATIHD + 4632048034; using two valid grammars for **ATD** and **ATI** (as shown in Figure 3) and invalid characters for which the target device returns ok but places a phone call to 4632048034 and shows the manufacturer, model revision, and IMEI information simultaneously.

(2) Syntactic errors – responds ok **with an action.** In this type of syntactically problematic grammars, the target device responds to an invalid command instance with ok but performs an action. For instance, the grammar **cmd** →**Arg1**. I .**Arg2** in Table 3 can be instantiated with an invalid command instance **ATHIX** which returns sensitive device information.

(3) Syntactic errors – responds error **with an action.** In this class of syntactic errors, the AT interface recognizes the inputs as faulty by acknowledging with error, but it still executes the action associated with the command and even does worse by crashing the RIL daemon and inducing complete disruptions in the cellular Internet connectivity. It basically reveals a fundamental flaw in the AT interface— *if a command is considered as erroneous, it should not be executed.* For example, the grammar **cmd** →D . **Dnum** in Table 3 can be instantiated with **ATD+4632048034** (a variation of the ATD production rule in Figure 3) which is supposed to start a cellular voice call. Instead, the grammar returns error in the form of NO CARRIER and induces the cellular Internet connectivity to go down completely for a certain amount of time (15-20 seconds). We have also found grammars for which the device returns other error statuses, e.g., ERROR, NO CARRIER, CME ERROR, ABORTED, and NOT SUPPORTED, but still executes those invalid commands.

(4) Semantic errors. This class of grammars conforms with the input pattern defined by the standards [11], but induces disruptions in the cellular connectivity for which the recovery requires rebooting the device. The grammars of this class are shown in Table 3.

Possible exploitation. It may appear that the implications of invalid AT commands over USB are negligible as compared to the valid AT commands which may wreak havoc by taking full control of the device. We, however, argue that if AT interface exposure is restricted through blacklisting the critical and unsafe valid AT commands by the parser in the first place, the adversary will still be able to induce the device to perform same semantic functionalities using invalid AT commands. This is due to the uncovered vulnerabilities for which the parser will fail to identify the invalid AT commands as the blacklisted commands and thus allows the adversary to achieve same functionalities as the valid ones.

5.4.1 *Efficacy of grammar-aware crossover (RQ3).*

ATFuzzer without crossover (by disabling the crossover in ATFuzzer) uncovered only 3 problematic grammars as compared to ATFuzzer with all proposed crossover and mutations (Table 4). This is due to the fact that ATFuzzer without crossover cannot induce enough changes in the structure and type of the arguments of parent grammars, as a result of which it reduces the search space.

5.4.2 *Efficacy of grammar-aware mutation (RQ4).*

Since ATFuzzer without mutation cannot induce changes in the arguments and the respective conditions, it uncovered only 2 problematic grammars. ATFuzzer without crossover, however, performs slightly better than that of the ATFuzzer without crossover. This also justifies our intuition that mutation strategies play a vital role in any fuzzer as compared to crossover techniques. Without mutation, a fuzzer unlikely generates interesting inputs for the system under test.

5.4.3 *Efficacy of timing feedback (RQ5).*

We observed that ATFuzzer without feedback performs better than the other two (RQ2 and RQ3) variations. ATFuzzer without feedback uncovered 5 problematic grammars and thus is less effective than ATFuzzer with feedback. AT interface being a complete black box with little to no feedback

Class of Bugs	Grammar and Command Instance	action/implication	Nexus5	LG G3	Nexus6	Nexus6P	HTC	S8plus
Syntatctic –returns OK with composite actions	**cmd** $\rightarrow$I.**Arg**.D.**Dnum**.**Darg**; **Arg** $\rightarrow [a-zA-Z]$ **Dnum** $\rightarrow [a-zA-Z0-9+*\#]^*$ **Darg** $\rightarrow I\|G\|\epsilon$ E.g. ATIHD + 4642048034I;	read, execution/ leaks manufacturer, model revision and IMEI	✓	✓	✓	✓	✓	
	cmd $\rightarrow$+COPN; **Arg** **Arg** $\rightarrow [i\|I]^*$ E.g. AT + COPN; III	read/ leaks list of operators, manufacturer, model revision and IMEI	✓	✓			✓	✓
Syntatctic – returns OK with an action	**cmd** $\rightarrow$**Arg1**.I.**Arg1** $\rightarrow [X\|H]$ E.g. ATHIX	read/ leaks manufacturer, model revision and IMEI	✓	✓	✓	✓	✓	
	cmd $\rightarrow$**Arg1**.I.**Arg1**.**Arg1** **Arg1** $\rightarrow X$ E.g. ATXIX	read/ leaks manufacturer, model revision and IMEI				✓	✓	✓
	cmd $\rightarrow$**Arg1**.**Arg2**.**Arg3** **Arg1** $\rightarrow +CIMI\mid I\mid +CEER$ **Arg2** $\rightarrow *\|;$ **Arg3** $\rightarrow$Q|Z E.g. AT + CIMI $*$ Q	read/ leaks IMSI, manufacturer, model revision and IMEI	✓	✓				
	cmd $\rightarrow$**+CLCC**;**Arg1** **Arg1** $\rightarrow [a-zA-Z0-9]^*$ E.g. AT + CLCC; ABC123	read/leaks current call list	✓	✓		✓	✓	✓
	cmd $\rightarrow$**Arg1**; **Arg2** **Arg1** $\rightarrow +COPN\| + CGMI\|$ $+CGMM\| + CGMR$ **Arg2** $\rightarrow [X\|E]$ E.g. AT + COPN; X	read/leaks list of operators, IMEI, model and revision information of the device	✓	✓	✓	✓	✓	✓
Syntatctic – returns ERROR with an action	**cmd** $\rightarrow$**Arg1**.**Arg2**.**Arg3** **Arg1** $\rightarrow +CIMI\mid I\mid +CEER$ **Arg2** $\rightarrow ;\|*$ **Arg3** $\rightarrow \hat{}[Q\|Z]$ E.g. ATI; L	read/ leaks IMSI, manufacturer, model revision and IMEI	✓	✓	✓		✓	
	cmd $\rightarrow$**Arg1**; **Arg2** **Arg1** $\rightarrow +COPN\| + CGMI\|$ $+CGMM\| + CGMR$ **Arg2** $\rightarrow \hat{}[X\|E]$ E.g. AT + CGMM; O	read /leaks list of operators, IMEI, model and revision information of the device	✓	✓	✓	✓		
	cmd $\rightarrow$**Arg**.D.**Dnum**.**Darg**; **Arg** $\rightarrow [a-zA-Z]$ **Dnum** $\rightarrow [a-zA-Z0-9+\#]^*$ **Darg** $\rightarrow I\|G\|\epsilon$ E.g. ATMD + 4632048034	crash/ internet connectivity disruption	✓					
	cmd $\rightarrow$+CUSD=,**String** **String** $\rightarrow [a-zA-Z0-9+*\#]^*$ E.g. AT + CUSD =, ABC	crash/ internet connection disruption			✓	✓		
Semantic – returns OK with an action	**cmd** $\rightarrow$**+CCFC**=**Arg1**,**Arg2**,**Arg3**, 145,32,**Arg4**,13,27 **Arg1** $\rightarrow [1-5]$ **Arg2** $\rightarrow [1-2]$ **Arg3** $\rightarrow [0-9]^*$ **Arg4** $\rightarrow [a-zA-z0-9]^*$ E.g. AT + CCFC = 3, 2, 732235, 145, 32, cA4{NYv, 13, 27	crash/ internet connectivity disruption	✓		✓	✓		
	cmd $\rightarrow$**+COPS** = 0,**Arg1**,**Arg2**,2 Arg1 $\rightarrow [0\|1]$ **Arg2** $\rightarrow [a-zA-z]^*$ E.g. AT + COPS = 0, 1, c19v6fC, 2	crash/internet connectivity disruption		✓				

Table 3: Summary of ATFuzzer**'s findings over USB.**

we had to resort to various creative ways including timing information to generate feedback score. However, This resorts to an upper bound for the coverage information and loosely dictates ATFuzzer.

5.4.4 Comparison with other state-of-the-art fuzzer (RQ6). We compare the effectiveness of ATFuzzer against AFL (American Fuzzy Lop) [59]. Since current versions of AFL require instrumenting the test programs, we implemented a modest string fuzzer that adopts five mutation strategies (*walking bit flips*, *walking byte flips*, *known integers*, *block deletion* and *block swapping*) employed by AFL and incorporated our proposed timing-based feedback loop to it. We evaluate this AFL variant with 80 different seeds (consisting of valid and invalid command instances of randomly chosen 40 different AT reference grammars).

Fuzzing Approach	Problematic Grammars
ATFuzzer	9
ATFuzzer w/o feedback	5
ATFuzzer w/o crossover	3
ATFuzzer w/o mutation	2
Modified AFL	2

Table 4: Result obtained with different fuzzing approaches on Nexus5 over a period of 3days for each approach.

Table 4 shows that the AFL variant uncovered 2 different problematic grammars whereas ATFuzzer uncovers 9 unique grammars after running for 3 days. Though we decided to compare our tool with AFL, which is the best choice we had as AFL is considered the state-of-the-art tool for fuzzing, we do not claim the comparison to be ideal. Because AFL relies heavily on code average information and for our case, we replaced the code coverage with the best available substitute (i.e., coarse-grained timing information as a loose indicator to code coverage). We acknowledge that this is a best-effort approach and the evaluation may be sub-optimal.

6 RELATED WORK

In this section, we mainly discuss the relevant work on the following four topics: AT commands, mutation-based fuzzing, grammar-based mutation, grammar-based generation.

AT commands. Most of the previous work related to AT commands follow investigate how an adversary can misuse valid AT commands to attack various systems. The work from Tian et al. [53] can be considered the most relevant to our work, however, it is significantly different in the following three aspects: (i) Firstly, they only show the impact of AT commands over USB as they consider the functionality and scope of AT commands over Bluetooth too limited to study. We, however, demonstrate the dire consequences of AT commands over Bluetooth interface with the uncovered invalid and valid AT commands. (ii) Secondly, they only show the impact of valid AT commands whereas we demonstrate the impact of invalid AT commands exploring different attack surfaces. (iii) Finally, one of the primary objectives of our work is to test the robustness of the AT interface, which is a different and complimentary end objective than theirs.

BlueBug [24] exploits a Bluetooth security loophole on few Bluetooth-enabled cell phones and issues AT commands via a covert channel. It, however, relies on the Bluetooth security loophole to attack and does not apply to all phones. In contrast, we have demonstrated a variety of attacks using valid and invalid AT commands running over Bluetooth which do not rely on any specific Bluetooth assumptions and also applicable to all the modern smartphones we had in our corpus. Injecting AT commands on android baseband was previously discussed on the XDA forum [23]. Pereira et al. [43, 45] used AT commands to flash malicious images on Samsung phones. Hay [29] discovered that AT interface can be exploited from Android bootloader and discovered new commands and attacks using the AT interface. AT commands have been used to exploit modems other than smartphones as well. Most prominently, USBswitcher [44, 49] and [43] demonstrate how these commands expose actions potentially causing security vulnerabilities in smartphones. Some other work use AT commands as a part of their tool, for instance, Mulliner et al. [42] use the AT commands as feedback while fuzzing SMS of phones. Xenakis et al. [57, 58] devise a tool using AT commands to steal sensitive information from baseband. None of them, however, actually analyzes or discovers bugs in the AT parser itself.

Mutation based fuzzers. Initial mutation-based fuzzers [41] used to mutate the test inputs randomly. To make this type of fuzzers more effective, a huge amount of work has been carried out to develop sophisticated techniques to improve mutation strategies— coverage information through instrumenting the binary [28, 36, 37, 59]; resource usage information [35, 46]; control and data flow features [48]; static vulnerability prediction models [38]; data-driven seed generation [55]; high-level structural representation of seed file [47]. There are also a few mutation-based fuzzers that incorporate the idea of grammars rather than inputs. Wang et al. [56] use grammars to guide mutation whereas Aschermann et al. [26] rely on code coverage feedback. Simulated annealing power schedule with genetic fuzzing has also been incorporated in [27]. However, due to the black-box nature of our system and structural pattern of AT command inputs, none of the existing concepts suffice fuzzing AT parser.

Generation-based fuzzers. Generation based fuzzers generate inputs based on a model [19, 20, 31, 54], specification or defined grammars. However, to the best of our knowledge, no fuzzer discovers a class of bugs at the grammar-level, rather generates concrete input instances. There are also some generation-based, more precisely, defined grammar-based fuzzers [16] [15] which use manually specified grammars as inputs. For instance, Mangeleme is an automated broken HTML generator and fuzzer, and Jsfunfuzz [15] uses specific knowledge about past and present vulnerabilities and uses grammar rules to produce inputs that may cause problems. Both of them are, however, random fuzzers.

7 DISCUSSION

Defenses. Our findings show that current implementations of baseband processors and AT command interfaces fail to correctly parse and filter out some of the possible anomalous inputs. In this paper, we do not explicitly explore defenses for preventing malicious users from exploiting these flaws. However, our findings signify that restricting the AT interface through access control policies, black-listing may not work due to the parsing bugs and invalid AT commands that the parser executes. Completely removing the exposure of AT modem interface over Bluetooth and USB can resolve the problem. Other than that, at a conceptual level, having a formal grammar specification of the supported AT command grammar may provide a better way to test the AT interface. Another aspect that particularly requires attention is the deployment of stricter policies that filter out anomalous AT commands.

Responsible disclosure. Given the sensitive nature of our findings, we have reported these to the relevant stakeholders (e.g., respective modems and devices vendors and manufacturers). Moreover, following the responsible disclosure policy, we have waited 90 days before making our findings public. Currently, to our knowledge, Samsung has been working to issue a patch to mitigate the vulnerabilities.

8 CONCLUSION AND FUTURE WORK

The paper proposes ATFuzzer for testing the correctness of the AT interface exposed by the baseband processor in a smartphone. Towards this goal, ATFuzzer leverages a grammar-guided evolutionary fuzzing-based approach. Unlike generational fuzzers which use the input grammar to generate syntactically correct inputs, ATFuzzer mutates the production rules in the grammar itself. Such an approach enables ATFuzzer to not only efficiently navigate the input search space but also allows it to exercise a diverse set of input AT commands. In our evaluation with ATFuzzer on 10 Android smartphones from 6 vendors revealed 4 invalid AT command grammars that are processed by the Bluetooth AT interface and can induce DoS, downgrade connectivity, and privacy leaks. For the USB AT interface, on the other hand, ATFuzzer identified 13 invalid AT command grammars which are equally damaging as the ones found for the Bluetooth AT interface. Our findings have been responsibly shared with the relevant stakeholders among which Samsung has acknowledged our findings and are working towards a patch. Two of our findings have also been assigned CVEs (CVE-2019-16400 and CVE-2019-16401).

Future work. In the future, we want to apply hybrid fuzzing in our problem domain. In the hybrid fuzzing paradigm, a black-box fuzzer's capabilities are enhanced through the use of lightweight static analysis (e.g., dynamic symbolic execution, taint analysis). Such an approach would, however, require us to address the issues concerning firmware binaries' practice of employing obfuscation and encryption.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their suggestions. This work is supported by NSF grants CNS-1657124 and CNS-1719369, Intel, and a grant by Purdue Research Foundation.

REFERENCES

[1] [n.d.]. AT Commands For CDMA Wireless Modems. http://www.canarysystems.com/nsupport/CDMA_AT_Commands.pdf.

[2] [n.d.]. CVE-2016-4030. https://nvd.nist.gov/vuln/detail/CVE-2016-4030.

[3] [n.d.]. CVE-2016-4031. https://nvd.nist.gov/vuln/detail/CVE-2016-4031.

[4] [n.d.]. CVE-2016-4032. https://nvd.nist.gov/vuln/detail/CVE-2016-4032.

[5] [n.d.]. CVE-2019-16400. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-16400.

[6] [n.d.]. CVE-2019-16401. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-16401.

[7] [n.d.]. CWE-325: Missing Required Cryptographic Step - CVE-2018-5383. In *Cernegie Mellon University ,CERT Coordination Center*. https://www.kb.cert.org/vuls/id/304725/.

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

[9] [n.d.]. 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). https://www.etsi.org/deliver/etsi_ts/127000_127099/127007/13.06.00_60/ts_127007v130600p.pdf.

[10] [n.d.]. Digital cellular telecommunications system (Phase 2+); Specification of the Subscriber Identity Module -Mobile Equipment (SIM-ME) interface (3GPP TS 51.011 version 4.15.0 Release 4). https://www.etsi.org/deliver/etsi_TS/151000_151099/151011/04.15.00_60/ts_151011v041500p.pdf.

[11] [n.d.]. Digital cellular telecommunications system (Phase 2+), Universal Mobile Telecommunications System UMTS, LTE, AT command set for User Equipment UE. https://www.etsi.org/deliver/etsi_ts/127000_127099/127007/10.03.00_60/ts_127007v100300p.pdf.

[12] [n.d.]. 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 5.3.0). https://www.etsi.org/deliver/etsi_gts/07/0705/05.03.00_60/gsmts_0705v050300p.pdf.

[13] [n.d.]. EVDO and CDMA AT Commands Reference Guide. https://www.multitech.com/documents/publications/manuals/s000546.pdf.

[14] [n.d.]. HUAWEI MU609 HSPA LGA Module Application Guide. https://www.paoli.cz/out/media/HUAWEI_MU609_HSPA_LGA_Module_Application_Guide_V100R002_02(1).pdf.

[15] [n.d.]. jsfunfuzz [online]. https://github.com/MozillaSecurity/funfuzz/tree/master/src/funfuzz/js/jsfunfuzz.

[16] [n.d.]. Mangleme [Online]. https://github.com/WebKit/webkit/tree/master/Tools/mangleme.

[17] [n.d.]. Motorola AT Command Set. https://ipfs.io/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco/wiki/Motorola_phone_AT_commands.html.

[18] [n.d.]. Neo 1973 and Neo FreeRunner GSM modem, AT Command set. http://wiki.openmoko.org/wiki/Neo_1973_and_Neo_FreeRunner_gsm_modem.

[19] [n.d.]. Peach Fuzzer Platform [online]. https://www.peach.tech/.

[20] [n.d.]. Radamsa [online]. https://gitlab.com/akihe/radamsa.

[21] [n.d.]. Sony Erricsson AT Command set. https://www.activexperts.com/sms-component/at/sonyericsson/.

[22] [n.d.]. Wikipedia. https://en.wikipedia.org/wiki/Hayes_command_set.

[23] [n.d.]. XDA Forum [online]. https://forum.xda-developers.com/galaxy-s2/help/how-to-talk-to-modem-commands-t1471241.

[24] M.Herfurt A. Laurie, M. Holtmann. [n.d.]. The bluebug. AL Digital Ltd. https://trifinite.org/trifinite_stuff_bluebug.html#introduction.

[25] Iosif Androulidakis. 2011. Intercepting mobile phone calls and short messages using a gsm tester. In *International Conference on Computer Networks*. Springer, 281–288.

[26] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[27] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2329–2344.

[28] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. [n.d.]. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, Vol. 00. 660–677. https://doi.org/10.1109/SP.2018.00040

[29] Roee Hay. 2017. fastboot oem vuln: android bootloader vulnerabilities in vendor customizations. In *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*.

[30] Roee Hay and Michael Goberman. 2017. Attacking Nexus 6 & 6P Custom Bootmodes. (2017). https://www.docdroid.net/dxKUj5c/attacking-nexus-6-6p-custom-bootmodes.pdf.

[31] Christian Holler, Kim Herzig, and Andreas Zeller. [n.d.]. Fuzzing with Code Fragments.

[32] Syed Rafiul Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. 2018. LTEInspector: A Systematic Approach for Adversarial Testing of 4G LTE. In *25th Annual Network and Distributed System Security Symposium, NDSS, San Diego, CA, USA, February 18-21*.

[33] Syed Rafiul Hussain, Mitziu Echeverria, Omar Chowdhury, Ninghui Li, and Elisa Bertino. 2019. Privacy Attacks to the 4G and 5G Cellular Paging Protocols Using Side Channel Information. (2019).

[34] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[35] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. ACM, New York, NY, USA, 254–265. https://doi.org/10.1145/3213846.3213874

[36] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 475–485. https://doi.org/10.1145/3238147.3238176

[37] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 627–637. https://doi.org/10.1145/3106237.3106295

[38] Yuwei Li, Shouling Ji, Chenyang Lv, Yuan Chen, Jianhai Chen, Qinchen Gu, and Chunming Wu. 2019. V-Fuzz: Vulnerability-Oriented Evolutionary Fuzzing. *CoRR* abs/1901.01142 (2019). arXiv:1901.01142 http://arxiv.org/abs/1901.01142
[39] Angela Lonzetta, Peter Cope, Joseph Campbell, Bassam Mohd, and Thaier Hayajneh. 2018. Security vulnerabilities in Bluetooth technology as used in IoT. *Journal of Sensor and Actuator Networks* 7, 3 (2018), 28.
[40] Ulrike Meyer and Susanne Wetzel. 2004. A man-in-the-middle attack on UMTS. In *Proceedings of the 3rd ACM workshop on Wireless security*. ACM, 90–97.
[41] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. https://doi.org/10.1145/96267.96279
[42] Collin Mulliner and Charlie Miller. 2009. Fuzzing the Phone in your Phone (Black Hat USA 2009).
[43] André Pereira, Manuel Correia, and Pedro Brandão. 2014. Charge your device with the latest malware.. In *BlackHat Europe*.
[44] André Pereira, Manuel Correia, and Pedro Brandão. 2014. USB Connection Vulnerabilities on Android Smartphones: Default and Vendors' Customizations. In *Communications and Multimedia Security*, Bart De Decker and André Zúquete (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 19–32.
[45] André Pereira, Manuel Correia, and Pedro Brandão. 2014. USB connection vulnerabilities on android smartphones: Default and vendorsâĂŹ customizations. In *IFIP International Conference on Communications and Multimedia Security*. Springer, 19–32.
[46] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. *CoRR* abs/1708.08437 (2017). arXiv:1708.08437 http://arxiv.org/abs/1708.08437
[47] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. 2018. Smart Greybox Fuzzing. *CoRR* abs/1811.09447 (2018). arXiv:1811.09447 http://arxiv.org/abs/1811.09447
[48] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.
[49] P. Roberto and F. Aristide. 2014. Modem interface exposed via USB.. In *BlackHat Europe*. https://github.com/ud2/advisories/tree/master/android/samsung/nocve-2016-0004.
[50] David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. [n.d.]. Breaking LTE on layer two.
[51] Mike Ryan. 2013. Bluetooth: With low energy comes low security. In *Presented as part of the 7th {USENIX} Workshop on Offensive Technologies*.
[52] Wireless Solutions Telit. [n.d.]. AT Commands Reference Guide. https://www.telit.com/wp-content/uploads/2017/09/Telit_AT_Commands_Reference_Guide_r24_B.pdf.
[53] Dave (Jing) Tian, Grant Hernandez, Joseph I. Choi, Vanessa Frost, Christie Raules, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, and Kevin R. B. Butler. 2018. ATtention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem. In *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD, 273–290. https://www.usenix.org/conference/usenixsecurity18/presentation/tian.
[54] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming. In *Computer Security – ESORICS 2016*, Ioannis Askoxylakis, Sotiris Ioannidis, Sokratis Katsikas, and Catherine Meadows (Eds.). Springer International Publishing, Cham, 581–601.
[55] J. Wang, B. Chen, L. Wei, and Y. Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. 579–594. https://doi.org/10.1109/SP.2017.23
[56] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2018. Superion: Grammar-Aware Greybox Fuzzing. *CoRR* abs/1812.01197 (2018). arXiv:1812.01197 http://arxiv.org/abs/1812.01197
[57] Christos Xenakis and Christoforos Ntantogian. 2015. Attacking the baseband modem of mobile phones to breach the users' privacy and network security. In *Cyber Conflict: Architectures in Cyberspace (CyCon), 2015 7th International Conference on*. IEEE, 231–244.
[58] Christos Xenakis, Christoforos Ntantogian, and Orestis Panos. 2016. (U) SimMonitor: A mobile application for security evaluation of cellular networks. *Computers & Security* 60 (2016), 62–78.
[59] M. Zalewski. [n.d.]. American fuzzy lop. [online]. http://lcamtuf.coredump.cx/afl/.

A APPENDIX

A.1 Target Devices Configuration.

In this section, we provide additional detailed information about the required set up for the devices we tested.

Some of the devices we tested expose their modem functionality by default and therefore required no additional configuration (also listed in Table 1). On the other hand, for the devices that do not expose any modem, it was necessary to root them and set a specific type of USB configuration. The USB configuration can be changed by setting *sys.usb.config* property. All the devices can be accessed through ADB (Android Debug Bridge) and Fastboot tools. With ADB it is possible to access the device's file system, reboot it in different modes, such as `bootloader` mode, rooting it, and finally change the device's properties directly with the command `setprop <property-name> <value>`. With fastboot, it is possible to operate the device in `bootloader` mode, install new partitions and change pre-boot settings required for rooting. For LG Nexus 5, we had to set *sys.usb.config* from the default "mnt,adb" to "diag,adb" through `adb shell`. This setting allows to access the phone in `diagnostic` mode and therefore to communicate with the AT command interface. For Motorola Nexus 6 and Huawei Nexus 6P, the USB configuration can be changed by first rebooting the phone in `bootloader` mode and then issuing the command "`fastboot oem bp-tools-on`" and "`fastboot oem enable-bp-tools`" to Nexus 6 and Nexus 6P, respectively as reported in [30]. After establishing serial communication with the device, it is possible to communicate with the smartphone through the AT interface.