AIRBUGCATCHER: Automated Wireless Reproduction of IoT Bugs

Guoqiang Hua
SUTD, Singapore

Matheus E. Garbelini
SUTD, Singapore

Sudipta Chattopadhyay
SUTD, Singapore

Abstract*—Fuzzing has been proven to be an effective tool to find implementation bugs in a range of wireless Internet of Things (IoT) devices such as smartphones, trackers, smart wearables, routers, etc. However, reliable and automated reproduction of vulnerabilities reported by over-the-air (OTA) fuzzing pipelines remains an open problem. While bug reproduction is crucial for troubleshooting and fixing of security flaws, it remains a challenge due to the non-deterministic nature of wireless devices. In this context, we present AIRBUGCATCHER, a hardware and protocol agnostic tool to automatically identify reliable OTA attack vectors and reproduce bugs in commercial-off-the-shelf (COTS) IoT devices. AIRBUGCATCHER aims to address two fundamental challenges during reproduction of vulnerabilities: Reproduction of bugs under the non-deterministic communication of wireless devices and resolution of ambiguities during the attack vector analysis of bugs within fuzzing logs. AIRBUGCATCHER accomplishes this by firstly analyzing packet traces and logs from an existing fuzzing pipeline and extracting a minimal set of fuzzing packets that might be responsible for triggering bugs in the target IoT device. Subsequently, AIRBUGCATCHER reliably reproduces bugs by generating several proof of concept (PoC) codes (test cases) and executing them against the target to validate the root cause of bugs. AIRBUGCATCHER has been evaluated against four COTS IoT devices employing wireless protocols such as 5G NR, Bluetooth and Wi-Fi. The results show that AIRBUGCATCHER can reproduce 90.4% (40/44) of bugs (crashes or hangs) extracted from fuzzing logs and generate PoC code that contains minimal attack vectors. For instance, AIRBUGCATCHER only generates up to three fuzzed packets (i.e., three attack vectors) from fuzzing logs that contain ≈47K fuzzed packets. Finally, we demonstrate that a standard replay-based approach (i.e., attempting to replay all packets from fuzzing logs) fail to reproduce most bugs (15 out of 16) due to the non-deterministic nature of wireless protocol implementations. Overall, we highlight that AIRBUGCATCHER offers a valuable addition to IoT fuzz testing pipelines by automating the process of OTA bug reproduction and empowering researchers and developers to identify and fix security flaws in IoT devices more efficiently.

1. Introduction

The security of wireless communication protocols is crucial for the success of internet-of-things (IoT) systems.

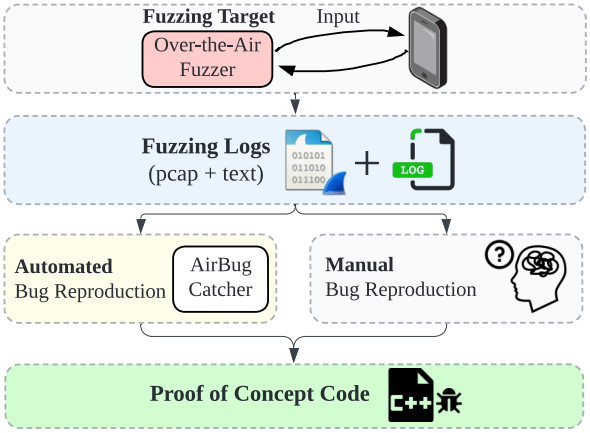

Figure 1: Illustration of bug reproduction workflow with and without AIRBUGCATCHER.

In the past few years, there has been a significant progress in designing offensive tools to automatically discover wireless protocol implementation vulnerabilities [21], [15], [10], [8], [5], [16]. Among others, over-the-air (OTA) wireless fuzzing has successfully uncovered numerous security vulnerabilities in Wi-Fi [5], Bluetooth Low Energy [15], [10], Bluetooth Classic or BT [8], LTE [16] and 5G NR [22], among others. While OTA fuzzing primarily focuses on the *discovery* of security vulnerabilities (e.g., crashes), in general, significant manual effort is involved to create minimal Proof of Concept (PoC) code that could *reliably reproduce the wireless vulnerabilities discovered by fuzzing*. This is critical to assist in the triaging and subsequent root-cause identification and patching of the vulnerability.

In this paper, we propose AIRBUGCATCHER, a systematic and automated process to reproduce wireless vulnerabilities on commercial-off-the-shelf (COTS) IoT devices. Figure 1 illustrates the position of AIRBUGCATCHER in bug reproduction workflow. In particular, based on the current fuzzing logs (i.e., packet traces and crash logs from the device), significant manual effort is involved to produce the PoC, which can be sent to the vendor for debugging. In AIRBUGCATCHER, we propose a systematic process that analyzes the fuzzing logs and automatically generates the PoC, thus substantially reducing the effort needed before patching.

In AIRBUGCATCHER, we address two key challenges in reliably reproducing a wireless vulnerability. Firstly, a

straightforward strategy would be to replay the exact sequence of benign and fuzzed packets that resulted in a vulnerability (e.g., a crash) during fuzzing campaign. While such a strategy may work for mostly deterministic protocols (e.g., TCP/IP, HTTP, FTP), this will often fail to reproduce vulnerabilities in wireless protocols that are inherently non-deterministic. Indeed, it is not possible to fully control the state of a wireless device during test, resulting a straightforward replay strategy impractical. AIRBUGCATCHER addresses this challenge by a fundamentally different method for bug reproduction. In particular, AIRBUGCATCHER first analyzes fuzzing logs using only a few rules and accurately computes conditions on targeted packet layers and field values that resulted the fuzzed packets leading to a bug. Subsequently, instead of *replaying* a test that contains previously recorded sequence of benign and fuzzed packets, AIRBUGCATCHER creates a test that acts as a *man-in-the-middle*. Concretely, such a test intercepts a packet only when the analyzed condition for fuzzed packet is met, then modifies targeted packet field values and send the modified packet towards the target IoT device on-the-fly. In such a fashion, AIRBUGCATCHER lets the wireless communication to proceed normally, except only for the targeted packets that need to be modified to reproduce a bug.

Secondly, even though fuzzing logs contain packet traces, such traces involve often hundreds or thousands of fuzzed packets, many of which are unrelated to the vulnerability (bug) under investigation. In practice, developer needs a minimal PoC that could help in the debugging of root cause. AIRBUGCATCHER addresses this via a two-stage process. Firstly, it analyzes fuzzing logs to group potentially identical bugs. Then, for each bug group, AIRBUGCATCHER conducts a systematic backward traversal on the packet traces, starting from the bug location. This heuristically computes many test scenarios, each of which contains a minimal set of fuzzed packets potentially related to the vulnerability. In the second stage, each such test scenario is leveraged for test code generation and reproduction in light of the process discussed in the preceding paragraph.

Prior works on automated PoC generation rely on intrusive approaches such as making use of external hardware [23] or through access to source code and firmware emulation [25]. In contrast, AIRBUGCATCHER provides the software security community with a tool that can be easily integrated to fuzzing pipelines quickly and non-intrusively. Moreover, the core process embodied within AIRBUGCATCHER is agnostic to the target hardware and protocol. In other words, if the wireless fuzzer supports the target fuzzing, then AIRBUGCATCHER can be coupled with such fuzzer to assist in triaging.

After providing a brief background and overview (Section 2), we present the following contributions:

1) We present the core methodology behind AIRBUGCATCHER, an automated process aimed at reliably reproducing wireless protocol vulnerabilities (Section 3).
2) We provide an open-source tool implementing the methodology behind AIRBUGCATCHER, which can be easily integrated with wireless fuzzing tools (Section 4).
3) We evaluate AIRBUGCATCHER with four IoT devices employing three different wireless protocols: 5G NR, Bluetooth Classic and Wi-Fi. Our evaluation reveals that such devices exhibit more than 240 bugs (crashes or hangs) in the fuzzing log. AIRBUGCATCHER first discovers that only 44 of these bugs are potentially unique. Subsequently, AIRBUGCATCHER automatically and reliably reproduces 40 bugs and confirms 33 of them are related to the bugs appearing in the fuzzing log. This shows the efficacy of AIRBUGCATCHER when paired to existing state-of-the-art wireless fuzzers [8], [22] (Section 5).
4) Our evaluation reveals that AIRBUGCATCHER generates minimal PoCs. The maximum number of mutation or replay actions in the PoC are limited to only three, whereas the respective fuzzing logs contain up to 46,992 mutations and up to 12,645 replay actions (Section 5).
5) We show that AIRBUGCATCHER is reasonably efficient. For example, in OnePlus phone, AIRBUGCATCHER reproduces and generates PoC for 13 unique crashes in $\approx$ two hours (Section 5).
6) We compare AIRBUGCATCHER with a replay-based approach and show that such a replay-based approach reproduces only one out of 16 bugs in an IoT device, whereas AIRBUGCATCHER reproduces 16 bugs.

After discussing some threats (Section 6) for our approach and related work (Section 7), we conclude in Section 8.

2. Background and Overview

In this section, we present an overview and example use-case of AIRBUGCATCHER in a glance.

Wireless Fuzzing: AIRBUGCATCHER is suitable for use with protocol software testing and particularly with over-the-air (OTA) fuzzing. Such category of fuzzing tools are centered around testing protocol implementation in a grey-box or blackbox fashion. In particular, the implementation of these fuzzing tools involves exchanging mutated or replayed inputs (packets) and responses over-the-air via an RF antenna, as opposed to a wired or internal loop-back interface. Example of state-of-the-art (SOTA) wireless fuzzers includes Sweyntooth [10], Braktooth [8], BLEDiff [15], Owfuzz [5], U-Fuzz [22] etc. These tools support testing the implementation of several Internet of Things (IoT) devices employing wireless protocol such as Bluetooth Low Energy, Bluetooth Classic, 5G NR etc. During the testing with such devices (i.e., fuzzing campaign), such tools expose logs containing communication traces of protocol packets exchanged during the fuzzing campaign and core dumps once a bug is triggered within the target. However, reproduction of bugs obtained during the fuzzing campaign is often manual

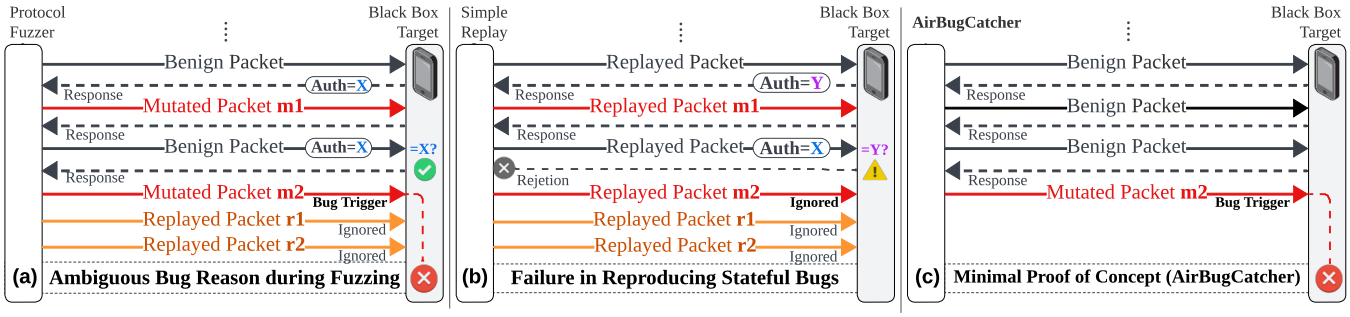

Figure 2: Example of fuzzed (i.e., mutated or replayed) packets that lead to (a) ambiguous bug reason (root cause) during fuzzing campaign; (b) failure in reproducing bugs due to simple replay not handling dynamic fields such as *Auth* (i.e., not using AIRBUGCATCHER); (c) minimal reproduction of bug via use of AIRBUGCATCHER.

and time consuming. In this context, AIRBUGCATCHER is designed to leverage fuzzing logs of an existent fuzzing pipeline such that bugs reported by an arbitrary wireless fuzzer are automatically reproduced via creation of PoC C++ codes that are ready to be sent to the software vendor for debugging and patching.

Motivation: Reproducing bugs in wireless protocols implementation is a highly challenging task as opposed to reproducing bugs in wired protocols. This is because a complete control of the testing environment is not possible when testing blackbox targets over-the-air. Firstly, repeating the same sequence of packets of the fuzzing campaign is not guaranteed to trigger bugs due to non-deterministic and stateful behaviour of the target's response [8], [22], [9]. Secondly, many mutated and replayed packets during the fuzzing campaign are often not related to causing a certain bug (e.g., crash). This results in ambiguity when analyzing the root cause of such a bug [8], [10].

For example, consider the fuzzing campaign illustrated in Figure 2(a) where several benign and fuzzed packets are sent to the target. Subsequently, the target crashes (e.g., segmentation fault) after receiving packet *r2* from the fuzzer. In such a case, the user is aware of the sequence of benign and fuzzed packets (i.e., mutated or replayed) before the bug is triggered. A simple process will attempt to replay this sequence as shown in Figure 2(b). However, the *third* replayed packet is rejected by the target, before *m2* could trigger the bug. This is because replayed packets (regardless if they are benign or not) do not preserve messages/fields that contain dynamic information such as authentication messages, features exchange in Bluetooth, and several 16-bytes authentication parameters for 5G NR (i.e., *Auth* parameter in Figure 2(b)). Therefore, replayed packets contain other fields of several bytes that are invalid in subsequent communication sessions and can be simply dropped by the target. This invalidates the attempted replay, as the target may not process the fuzzed message down to the specific mutated bytes.

More broadly, failure in reproducing wireless bugs during replay is a common phenomenon due to the non-deterministic nature of wireless protocols. Intuitively, this happens due to the wireless target replying differently (e.g., unsolicited requests, different message order/response etc.) such that **long** replay sequences get broken and not due to any false positives during the fuzzing campaigns. While the target internal state and protocol replies can be enforced in source code or emulation-based fuzzing, this level of control is infeasible for over-the-air fuzzing with closed source wireless stacks. Figure 2(a) also shows multiple mutated and replayed packets during fuzzing campaign. However, it is the packet *m2*, which results in the bug, irrespective of what packets are exchanged afterwards. Hence, simply replaying packet *r2* or *r1*, which are indeed the packets closer to the bug location in the packet trace, results in an unsuccessful reproduction of the bug. In reality, even if there are not too many bugs, number of fuzzed messages, during the fuzzing campaign, could be significant (often in the order of several thousands). Thus, manually identifying minimal set of fuzzed messages causing the bug (typically 1-3, as shown in our evaluation) is still infeasible.

To address the challenges mentioned in the preceding paragraph, AIRBUGCATCHER (illustrated in Figure 2 (c)) aims to significantly reduce the time taken to manually analyze and reproduce bugs. This is accomplished by automatically creating test cases with the minimal amount of fuzzing actions (i.e., mutation or replay), while avoiding the ambiguity to reason about crashes during bug reproduction.

AIRBUGCATCHER Workflow: AIRBUGCATCHER offloads the task of reproducing bugs discovered by the fuzzer (i.e., manual reproduction) to the pipeline presented in Figure 3. Firstly, in a post-fuzzing scenario (Step 1), packets traces (i.e., PCAP file) and target logs (i.e., crash/segmentation-fault dump), as captured during the fuzzing campaign, are fed to AIRBUGCATCHER for further analysis. The target logs are usually captured via serial port or by using standard tools such as *Logcat* for Android smartphones targets. Concurrently, the communication between the fuzzer and the target is monitored during the fuzzing campaign and recorded into the standard *packets trace* format such as *PCAP*.

Subsequently, the component *Packet Analysis* (Step 2) receives both the packets trace and target crash log. The aim of this component is to accurately identify the attack vector (e.g., the minimal set of mutated and replayed packets) for

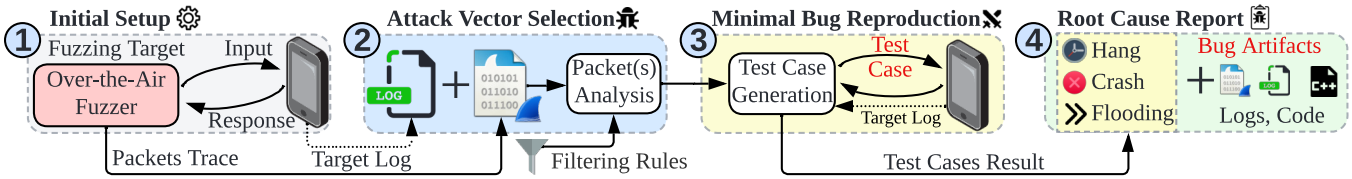

Figure 3: Overview of AIRBUGCATCHER

each unique crash resulting in the packet trace. To this end, AIRBUGCATCHER first conducts a simple analysis to identify and group the fuzzed packets related to each potentially unique crashes. Then, the Packet Analysis (Step ②) component systematically selects fuzzed packets within such groups based on a tunable sliding window that moves backwards from the crash location (Crash indication in Step ①). The outcome of this step results in many test scenarios, each with minimal set of selected fuzzed packets. Such test scenarios are sent to the component *Test Case Generation* in Step ③ for test-code generation, execution and validation of the respective crash.

For each test scenario computed in Step ②, the component *Test Case Generation* first generates the respective test code. Such is accomplished by analyzing the fuzzed packets respective to the test scenario and translating the fuzzing action (e.g., mutation or replay) into C++ code. Once the test code is generated, it is compiled into binary code and run against the same target of step ①. During the execution of such test cases, packets trace and target logs are saved and used to determine whether bugs analyzed from step ② are reproduced correctly. This step is repeated until all bugs found by the fuzzer are evaluated or a time budget is reached. The outcome of ③ is a report (step ④), which aggregates all the relevant bug information (e.g., type of bug such as crash, hang or flooding) and a minimal PoC code (i.e., the test case from Step ③) for each potentially unique bug identified in Step ② and reproduced in Step ③. Such a report can then be sent to the vendor affected by the bugs in order to accelerate the triaging and fixing process.

Collection of fuzzing logs: AIRBUGCATCHER can easily work with existent OTA fuzzers by (i) reusing packet traces collected during the fuzzing campaign and (ii) supporting collection of target logs via serial port, *ssh*, *Logcat* or other debugging tools capable of exporting logs to text files. Such a non-intrusive approach makes AIRBUGCATCHER easily adoptable by the software security community.

3. Methodology

At a high level, the design of AIRBUGCATCHER is separated into two distinct stages. The first stage performs ***Offline Bug Analysis*** (Step ② of Figure 3) of fuzzing logs and extracts two relevant outputs: (i) groups of bugs with the same bug identifier and (ii) a set of test scenarios that can potentially trigger corresponding bug groups within the target. The second stage of AIRBUGCATCHER, leverages the set of test scenarios to perform **Over-the-Air Bug Reproduction** against the target (Step ③ of Figure 3). This is performed

TABLE 1: AIRBUGCATCHER parameters used in Offline Bug Analysis and Over-the-Air Bug Reproduction

Parameter	Meaning	Default Value
C_t	Crash Timeout	1 minute
H_t	Hang Timeout	2 minutes
F_t	Flood Timeout	2 minutes
Max_{fpg}	Maximum Fuzzed Packet Generation	3
Max_{tt}	Maximum Trial Time	1 hour
Max_{lfi}	Maximum Lookback Fuzzed Iteration	3
F_e	Flooding Enabled	true
M_e	Mutation Enabled	true
R_e	Replay Enabled	true

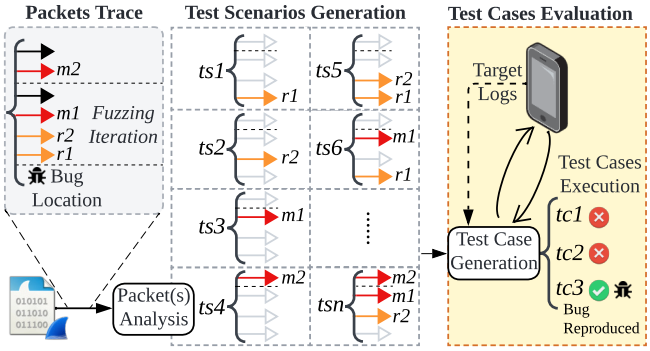

Figure 4: Illustration of AIRBUGCATCHER Packet Analysis and subsequent validation of test scenarios (ts). Labels $r1, r2$ indicate replayed packets (highlighted in yellow) and labels $m1, m2$ indicate mutated packets (highlighted in red).

repeatedly until all bug groups are evaluated and categorized or a time budget is reached. In the following subsections, we describe the design and implementation details of each AIRBUGCATCHER stage.

3.1. Offline Bug Analysis

Fuzzed Packet & Bug Discovery: Initially, AIRBUGCATCHER takes fuzzing logs (i.e., *Packet Trace* and *Target Logs*) and packet filtering rules as inputs from user. These packet filtering rules are used to identify (fuzzed) packet states and they borrow a syntax alike Wireshark display filters. Intuitively, such rules specify the names of packet fields which hold values of packet types (states) for different protocol layers. In Section 3.2, we show an example of such rules, whereas all rules used for the evaluated protocols are provided in Appendix A. It is worthwhile to mention that even though packet filtering rules need to be supplied manually, this does not involve a significant manual effort. This is because a security analyst is usually aware of the parts of the protocol being fuzzed. AIRBUGCATCHER starts

the offline analysis by traversing the packets trace within the provided fuzzing log to chronologically discover all fuzzed packets (i.e., mutated or replayed packets) and bug locations. Figure 4 illustrates a packet trace containing the aforementioned mutated and replayed packets as arrows, most of which belong to the same *fuzzing iteration* $(r1, r2, m1)$.

Each *fuzzed packet* (i.e., *fuzzed_packet*) is processed and stored as an object consisting of raw bytes and additional attributes such as *packet state*, *packet filter*, and *fuzzing type* (classified as either *mutation* or *replay*). Attributes such as packet state and packet filter are specially introduced by AIRBUGCATCHER to help identify specific packets during communication via stateful protocols such as 5G NR, Bluetooth and Wi-Fi. These attributes are generated using packet filtering rules, as mentioned in the preceding paragraph. Notably, the packet filter captures a logical condition to precisely categorize the packets belonging to the respective packet state. Thus, such packet filters are used subsequently during our test code generation to intercept and modify packets. In summary, for each bug in the fuzzing log, *fuzzed_packet* contains a list of all fuzzed packets (and their attributes) encountered between the moment the bug appears in the packet traces and the moment when the previous bug appears or the start of fuzzing campaign, whichever is earlier. In addition, we classify the type of a bug: *crash* or *hang*. While a *crash* is indicated in the packet traces or target logs, *hangs* are identified via the use of a timeout. This is due to large delays in target's consecutive responses during hang.

Bug Identifier and Bug Grouping: In the fuzzing log, numerous bugs appear to be triggered by the same root cause. Nonetheless, such bugs often manifest as separate ones in the fuzzing logs. To help reduce the number of bugs for AIRBUGCATCHER to reproduce, we group bugs via the use of *bug identifiers*. This bug identifier is obtained by analyzing both the packets trace and target logs. In particular, we identify bug patterns based on their types: *(i)* For *crashes*, AIRBUGCATCHER identifies crash dump or reboot messages in the target logs associated with the bug location. *(ii)* Concurrently, *hangs* are identified by searching for the unresponsiveness of the target within a certain amount of time. When the target logs emit useful information (e.g., source-code line and/or memory addresses) upon crashes, we identify and group crashes based on such information. In the case that no target logs are available for analysis or such logs do not provide additional information (e.g., source-code line or memory addresses), AIRBUGCATCHER constructs bug identifiers from packet state and groups bugs based on identical packet states. For crashes, we compute the packet state when the crash is manifested. For hangs, we posit that the last fuzzed packet in the fuzzing iteration may lead the target device to hang. Therefore, the bug identifier for such cases are captured using the packet state of the *closest fuzzed packet* before the hang. For example, such an identifier may be "*hang_TX / LMP / LMP_features_req*", where *hang* indicates the bug type, "*TX*" captures packet direction (transmission), "*LMP*" captures the packet protocol-layer and "*LMP_features_req*" is the packet type. More implementation-specific details of identifying and grouping bugs for a variety of target logs, is provided in Section 3.3.

Algorithm 1 Test Scenario Generation

```
1:  Input: Set of Bug Groups B_gs
2:  Output: Set of Test Scenarios TS[G] for each G ∈ B_gs
3:  ▷ Initialize the set of test scenarios for each bug group
4:  TS[G] := ∅ for each G ∈ B_gs
5:  for each G_id ∈ B_gs do
6:      ▷ Iterate over each bug within bug groups
7:      for each bug matching the identifier G_id do
8:          ▷ Initialize fuzzed packets and their combinations
9:          F_pkts ← ∅, C_pkts ← ∅
10:         ▷ Iterate over all packets associated with a bug
11:         for each pkt ∈ bug.fuzzed_packet do
12:             Let iter be the number of the fuzzing iteration of
                the packet
13:             if iter > Max_lfi then
14:                 break
15:             ▷ Store fuzzed packets associated with each bug
16:             if (M_e ∧ pkt.fuzzed_type ="mutation" )
                ∨ (R_e ∧ pkt.fuzzed_type ="replay") then
17:                 F_pkts ← F_pkts ∪ {pkt}
18:         ▷ Generate combinations of test scenarios in TS
19:         n := Max_fpg
20:         C_pkts ← ⋃_{i=1}^{n} combinations(F_pkts, i)
21:         TS[G_id] ← TS[G_id] ∪ {C_pkts}
22: return TS
```

Test Scenario Generation: The objective of this step is to extract sequences of fuzzed packets (i.e., test scenarios) corresponding to each bug group. Such a test scenario, once translated into a test case, will assist in the deterministic reproduction of the fuzzed packet sequence during over-the-air communication with the target. An example of test scenarios generation is shown in Figure 4. After processing and analyzing the packet trace, the resulting test scenarios are represented by $ts1$ to tsn, each containing a sequence of fuzzed packets that might be mutated $(m1, m2)$ or replayed $(r1, r2)$ towards the target.

Algorithm 1 illustrates the process to generate test scenarios. It receives as input the bug groups B_{gs} from the previous *Bug Grouping* step. Recall from Section 3.1: **Fuzzed Packet & Bug Discovery** that AIRBUGCATCHER stores a sequence of fuzzed packets (*fuzzed_packet*) potentially related to each bug. In Algorithm 1, we perform a bounded backward traversal on this sequence (i.e., *bug.fuzzed_packet* in line 11). The bounded traversal is controlled by the parameter Max_{lfi} (line 13). This intuitively captures the maximum amount of past fuzzing iterations to search for. Our intuition behind the backward search is based on a hypothesis that often packets closer (even though not the closest) to the bug location are responsible for the bug. The backward bounded search reveals a limited set of fuzzed packets F_{pkts} (line 17), which are then used for test scenario generation. In particular, the set of test scenarios from F_{pkts} are generated using all possible combinations of fuzzed packets up to a length Max_{fpg} (line 20). Even though such

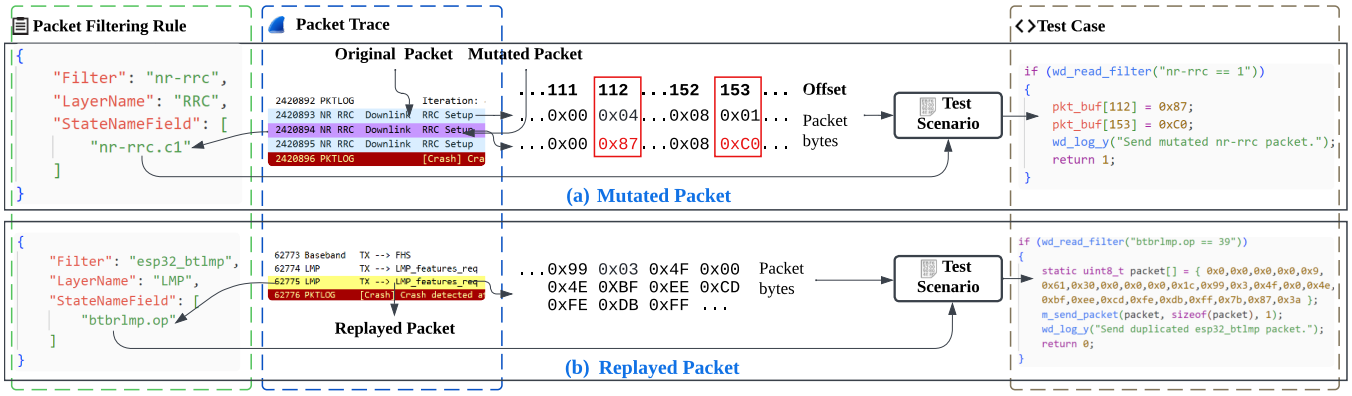

Figure 5: Illustration of test case generation for (a) Mutated Packet and (b) Replayed Packet in AIRBUGCATCHER.

a process may lead to a combinatorial explosion in theory, we observed that often a small value of Max_{fpg} (e.g., two-three) suffices in practice. Hence, our approach efficiently reproduces real world wireless bugs.

The outcome of Algorithm 1 is TS, the set of all test scenarios for each bug. It is worthwhile to mention that although the full set of test scenarios TS is sent to the Test Case Generation component, not all scenarios matching a group G_{id} needs to be evaluated. For example, test case is generated only for a representative bug in each group. Moreover, additional time budget is employed during Test Case Generation to better manage the efficiency of AIRBUGCATCHER for generating and executing test cases.

3.2. Over-the-Air Bug Reproduction

In this stage, AIRBUGCATCHER leverages the information computed by offline analysis, then generates and executes test code on IoT devices. The execution of the test code is also monitored to check whether the bugs are expected. In the following, we discuss these steps in detail.

Test Case Generation: This component of AIRBUGCATCHER takes a test scenario and translates it into a test case (code). We note that each test scenario is essentially a small set of fuzzed packets, as extracted from fuzzing logs. Moreover, the wireless fuzzing tools used by AIRBUGCATCHER approach [8], [22] record the corresponding original packet for each fuzzed packet transmitted. Using the set of fuzzed packets and their corresponding original packets, we transform the test scenario into a test case.

Figure 5 illustrates examples of test code generation. While Figure 5(a) illustrates the case for mutated packets, Figure 5(b) shows it for replayed packets. The leftmost side of Figure 5(a) shows an excerpt of the packet filtering rules (5G NR) used for extracting packet states. Concretely, the entry *Filter* shows the name of the protocol layer, whereas the entry *StateNameField* captures the name of the packet field holding the packet type. Thus, using the value of the field `nr-rrc.c1`, we can identify types of stateful RRC-layer packets. Figure 5 also illustrates the fuzzed packet (say P_μ) for the test scenario, the corresponding original packet (say P_O) and their raw bytes aligned with the byte offset.

Intuitively, the test case translation involves two steps: *(i)* computing the condition to selectively intercept packet for fuzzing, and *(ii)* computing the code to generate the fuzzed packet. In Figure 5(a), using the packet filtering rules, we extract the value of the field `nr-rrc.c1` from P_O and construct the filtering condition "`nr-rrc.c1=1`". Intuitively, this means that the test case will only fuzz packets when the condition "`nr-rrc.c1=1`" holds. After the creation of the condition, we generate the code for fuzzing the packet. For malformed packets, such is accomplished by comparing the raw bytes of P_μ and P_O. In Figure 5(a), P_μ differs from P_O in offsets 112 and 153. The modification of raw byte content in such offsets are accordingly generated (see the "Test Case" in Figure 5).

For replayed packets (see Figure 5(b)), the test case generation similarly involves the identification of condition "`btbrlmp.op = 39`" to select packets for fuzzing. However, in contrast to modifying the packet content, the same packet is sent twice whenever the filtering condition is met. This is accomplished via `m_send_packet`, which takes the raw packet bytes (`packet`), and replays the packet as many times as specified via a counter (1). Moreover, recall that our offline packet analysis associates a list of fuzzed packets with each bug discovered. We generate flooding test case only if the proportion of replayed packets in this list is larger than a threshold (set to 0.8 in our evaluation). This is similar to the test case for replay packets, except that the replayed packet is continuously sent to the target device (via `m_send_packet` interface) in a flooding test case.

It is worthwhile to mention that our test case does not replay previously recorded packet sequence. Instead, it selectively filters packets based on the analysis of fuzzing logs and modifies them accordingly during live communication. This approach inherently overcomes the difficulty and uncertainty of non-deterministic wireless protocols in the process of reproducing bugs. This is because the dynamic context of wireless communication is preserved. We also note that the test code generation process illustrated in Figure 5 is protocol agnostic and is applicable to bugs resulting due to arbitrarily mutated, replayed or flooded packet sequence.

Test Case Execution: AIRBUGCATCHER generates and

executes test cases sequentially for each group within a maximum time budget Max_{tt}. For each test execution, AIRBUGCATCHER monitors the logs and stops the test execution once an expected bug is triggered or a timeout set for bug detection (e.g., C_t, H_t and F_t in Table 1) is reached. The default timeout values for detecting hangs (H_t) and flooding (F_t) in our experiments are larger than the timeout set for crashes (C_t). This is because hangs and flooding typically require longer time to trigger the respective bugs.

Once a bug is triggered during test execution (either detected in the log or via the timeout parameters), the identifier of the triggered bug is compared against the identifier of the bug group under test. This is accomplished via the same methodology described in Section 3.1: **Bug Identifier and Bug Grouping**. When identifiers of the triggered bug and the bug group match, the test execution concludes that the expected bug identifier has been found for the bug group and AIRBUGCATCHER proceeds to execute the test cases in the next bug group. Figure 4 illustrates the execution process of test cases. Specifically, Figure 4 shows that AIRBUGCATCHER continues to the next bug group after the test case $tc3$ finds the expected bug. If no expected bug is found for a bug group after maximum trial time Max_{tt}, we abort the reproduction process for the current bug group and AIRBUGCATCHER moves on to generate and execute test cases for the next bug group.

3.3. Implementation Details

AIRBUGCATCHER is designed with generalizability in its focus. In particular, AIRBUGCATCHER is protocol agnostic and it is designed to work with different types of bugs (i.e., crashes and hangs). Moreover, AIRBUGCATCHER works with a variety of target wireless devices with different device log formats. Notably, among the evaluated devices (see Table 2), the SIMCom device does not output any logs, whereas Cypress device does not output any memory address/source-code line upon a crash. On the contrary, both the ESP32 device and the OnePlus phone emit source-code line upon a crash. In certain cases, the ESP32 device also outputs a memory trace. Finally, we note that hangs do not exhibit any visible output in device log due to unresponsive target. Therefore, hangs are identified via the state of the the fuzzed packet closest to the bug (see Section 3.1).

Implementation of Bug Identifier: While the design of AIRBUGCATCHER is general, we implement several strategies to deal with the device-specific log formats, when available. In particular, we developed bug identification strategies (see Section 3: **Bug Identifier and Grouping**) that work both with the presence and absence of target logs, albeit with different effectiveness. For instance, target devices such as ESP32-WROOM-32 and OnePlus phone (see Table 2) easily output detailed logs in the event of crashes, while other target devices such as Cypress Board do not output logs. In target that expose logs, there are two kinds of traces exposed during the moment a crash is triggered: source code line trace showing reachable assert e.g., "*ASSERT_PARAM(1 0), in ld_acl.c at line 1772*" and memory trace like "*0x40082dca:0x3ffbe2d0*". Furthermore, in the case that target logs are not available, the identifier is constructed from the packet state when the target crashes. We note that more precise identifiers are possible to construct, for example, by looking at a history of packet states. However, we only take the crashed location to keep the implementation of AIRBUGCATCHER simple and we show in our evaluation that such a strategy works well in practice.

Implementation of Bug Grouping: Once the bug identifiers are computed, they are used to group potentially identical bugs. While *hangs* can be grouped simply based on the associated identifier (which is typically the closest packet state before hang), the grouping process for *crashes* is slightly more involved. Specifically, when target logs show reachable assert (or similar) in the source code, crashes are grouped together if source-code line traces match exactly. Otherwise, memory trace is used to decide if bugs can be grouped. However, we noted that memory traces originated from certain IoT device crashes (e.g., in ESP32-WROOM-32 or ESP-WROVER-KIT) may manifest in a slightly different manner. Despite this, memory offset patterns exist in such traces, hence helping to group multiple crashes. For example, the following two bug identifiers extracted from ESP32-WROOM-32 fuzzing log (backtraces), are triggered by the same root cause (truncated for illustration):

```
BugID1=0x40101311:0x3ffcc170
       0x4001a637:0x3ffcc190...
BugID2=0x40101311:0x3ffcc580
       0x4001a637:0x3ffcc5a0...
```

There are two pairs of memory addresses in each *BugID* that are separated by a colon. The first addresses in both pairs of addresses in *BugID1* matches with that of *BugID2*. This, however, does not hold for the second addresses in both pairs. Nonetheless, the second addresses in both the pairs of addresses in the two bug identifiers are separated by identical offset. Due to such simple patterns, AIRBUGCATCHER groups both *BugID1* and *BugID2* together. Finally, if target logs are not available, the packet state of the bug location is used as the bug identifier. Consequently, only bugs triggered at the same packet state are grouped. All the bug groups discovered in this process are passed to the next step of AIRBUGCATCHER (Section 3: **Test Scenario Generation**).

False Positives and Negatives in Bug Grouping: We note that both false positives (i.e., the same bug with different identifiers) and negatives (different bugs with the same identifier) are possible in our bug grouping. This inaccuracy stems from the fact that AIRBUGCATCHER deals with the closed source firmware. Thus, AIRBUGCATCHER aims to leverage as much information (e.g., protocol states and target logs) as a human expert would have done in the absence of source code, to analyze and group the bugs. We believe the confirmation of truly unique bugs is only possible with the availability of the source code and such is unavailable in the case of over-the-air fuzzing. In general, AIRBUGCATCHER aims to speed up the triaging process with the device vendor.

TABLE 2: Target devices used in evaluation. Application name is not applicable (N.A.) on devices that do not require a specific application.

Protocol	Vendor	Target Device	Firmware	App. Name
BT	Espressif	ESP32-WROOM-32	ESP-IDF commit 3de8b79	bt_spp_acceptor
	Cypress	CYW920735Q60EVB-01	WICED SDK 2.9.0	rfcomm_serial_port
5G	OnePlus	OnePlus Nord CE 2	IV2201_11_F.48	N.A.
	SIMCom	SIMCom SIM8202G-M.2	SIM8202G-M2_V1.2	N.A.
WIFI	Espressif	ESP-WROVER-KIT	ESP-IDF commit b886dc6	wifi_enterprise

TABLE 3: Statistics Obtained from Target Fuzzing Logs.

Target Device	# Mutations	# Replays	# Packets	# Crashes	Duration
ESP32-WROOM-32	7860	12645	1,627,30	190	15 hr 46 min
CYW920735Q60EVB-01	2083	2691	45,769	12	17 hr 10 min
OnePlus Nord CE 2	46992	0	5,486,869	30	13 hr 17 min
SIMCom SIM8202G-M.2	9867	0	1,679,646	5	9 hr 58 min
ESP-WROVER-KIT	2549	1837	108,954	5	1 hr 25 min

Once the vendor triages the bug report, ambiguities in bug grouping (e.g., the same bug with different identifiers and PoCs) can be quickly verified due to the reproducible PoCs. Then we can tweak our bug grouping and run AIRBUGCATCHER again to reproduce the bugs that might have been incorrectly grouped and generate bug reports automatically.

4. Evaluation Setup

Hardware and Software Setup: We evaluated AIRBUGCATCHER on three different wireless protocols: BT (i.e., Bluetooth Classic), 5G NR and Wi-Fi. For each protocol, our evaluation includes one or two test devices (targets) to demonstrate that AIRBUGCATCHER works with different protocols and test devices. Table 2 shows the details of the evaluated devices, including vendor names, targeted protocols, firmware versions and application names. The firmware versions in Table 2 are chosen based on the fact that such versions were demonstrated to exhibit crashes in earlier works [22], [8]. To launch over-the-air (OTA) fuzzing campaign on the diverse set of devices employing multitudes of wireless protocols, we leveraged several off-the-shelf hardware devices. Specifically, we used *ESP32-Ethernet-Kit* and *ESP-WROVER-KIT* to establish a BT connection with the target ESP-WROOM-32 [8]. Both these devices are flashed with Braktooth firmware [8]. For 5G NR connectivity with smartphones (OnePlus), we used a *Software Defined Radio (SDR)* i.e., USRP B210 to create the 5G base station. Finally, an ALFA *AWUS036AC* dongle serves as the Wi-Fi access point (AP) in our evaluation. In addition, we use a USB Per-Port Control Hub to perform programmable power-cycles. This is to address scenarios where targets become unresponsive during the fuzzing campaign and require power cycles for the campaign to continue without any manual effort. Finally, we run both the fuzzing campaign and AIRBUGCATCHER workflow (see Figure 3) on a Beelink SER5 Mini PC with an AMD Ryzen 7 5800H processor and 12 GB memory, running Armbian 23.02.2 operating system with kernel 5.15.0 version.

AIRBUGCATCHER software is written in Python using 2710 lines of code (LoC). This includes fuzzing log analyzer and crash finder to locate crashes in the fuzzing logs (Step ② in Figure 3) as well as PoC (i.e., *test case*) generator and PoC runner (Step ③ in Figure 3).

Fuzzing Log Generation: To generate fuzzing logs that constitute the inputs of AIRBUGCATCHER, we use earlier works on BT fuzzing [8] and follow its default setup to launch OTA fuzzing on ESP-WROOM-32, Cypress board and ESP-WROVER-KIT. To run fuzzing on OnePlus phone and SIM8202G, we take advantage of open-source OTA fuzzing framework U-Fuzz [22] and follow its default setup to conduct 5G NR fuzzing. Table 3 presents statistics of the fuzzing logs from different test devices. The *#Mutation* and *#Replay* columns present the total number of mutated and replayed packets throughout the entire fuzzing campaign, respectively. In Table 3, we also report the total number of crashes in the fuzzing log. We note that fuzzing log generation is not the focus of AIRBUGCATCHER, instead AIRBUGCATCHER complements the security testing pipeline by leveraging the fuzzing logs to automatically create the PoC (Figure 3).

5. Evaluation Results

We address the following research questions to evaluate and demonstrate the capabilities of AIRBUGCATCHER:

RQ1: How effective is AIRBUGCATCHER to reproduce crashes? In order to evaluate the effectiveness of our experiments, we keep most of the AIRBUGCATCHER parameters unaltered from the default values of Table 1 and modify them only for certain devices. Table 4 outlines the effectiveness of AIRBUGCATCHER in reproducing crashes on different test devices. Firstly, for test with 5G devices such as OnePlus phone and SIM8202G, the fuzzing logs do not exhibit replayed packets. Consequently, parameters F_e and R_e are set to *false*. Secondly, our Wi-Fi test device ESP-WROVER-KIT disconnects from the Braktooth Wi-Fi Fuzzer in every fuzzing iteration. This means that it is sufficient to generate PoCs based on the fuzzed packets that are one iteration before a bug is triggered. Hence, Max_{lfi} is set to one (1).

The results of our evaluation are showcased in Table 4. When AIRBUGCATCHER tries to reproduce one *bug group*, it is possible that no bug is triggered at all. We categorize such results as **Not reproduced**. In contrast, a bug group is only categorized as **Reproduced** if AIRBUGCATCHER *Minimal Bug Reproduction* (see Figure 3) successfully triggers any bug within the target. However, there is a chance that the identifier of a triggered bug does not match the expected bug identifier (as analyzed by the Packet Analysis component) during test execution. Therefore, such cases are categorized as **Unexpected**. In this context, the column **Expected** indicates bugs that are successfully triggered with the expected bug identifier. For all cases, the results are partitioned in types of bugs (i.e., crash and hang) and quantified within the parenthesis. Last but not least, the *Max #Mutation* and *Max #Replay* columns represent the maximum number of mutated and replayed packets within each *generated PoC* across all trials of the *Test Case Generation*. Finally, the total number of generated PoCs and total time taken for running AIRBUGCATCHER are shown in columns *#Test Case* and *Time* respectively.

TABLE 4: Summary of results obtained by AIRBUGCATCHER. #Test Case denotes the total number of test cases generated with the aim to reproduce bugs from the respective fuzzing log.

Device	# Unique Bugs (# Crashes + # Hangs)	# Bugs (# Crashes + # Hangs)				Test Case Generation		# Test Case	Time
		Reproduced	Not reproduced	Expected	Unexpected	Max # Mutation	Max # Replay		
ESP32-WROOM-32	16 (14 + 2)	16 (14 + 2)	0	11 (9 + 2)	5 (5 + 0)	3	2	332	6 hr 44 min
Cypress Board	6 (4 + 2)	5 (3 + 2)	1 (1 + 0)	4 (3 + 1)	1 (0 + 1)	2	2	46	1 hr 26 min
OnePlus Phone	14 (13 + 1)	13 (13 + 0)	1 (0 + 1)	13 (13 + 0)	0	3	0	82	2 hr 09 min
SIM8202G	4 (4 + 0)	4 (4 + 0)	0	3 (3 + 0)	1 (1 + 0)	3	0	54	1 hr 26 min
ESP-WROVER-KIT	4 (4 + 0)	2 (2 + 0)	2 (2 + 0)	2 (2 + 0)	0	3	3	126	2 hr 53 min
Total (All Devices)	**44 (39 + 5)**	**40 (36 + 4)**	**4 (3 + 1)**	**33 (30 + 3)**	**7 (6 + 1)**	—	—	**640**	**14 hr 38 min**

Our findings reveal that AIRBUGCATCHER triggers all the 16 bugs within BT target ESP32-WROOM-32, while 11 out of 16 bug groups have expected reproductions. On the contrary, there are only two reproductions out of 4 unique bugs for Wi-Fi target ESP-WROVER-KIT, while the two reproductions are also expected. This is possibly because the Bluetooth stack used in ESP-WROVER-KIT involves many states and complex protocol procedures. In regards to the time to complete all trials, ESP32-WROOM-32 takes 6 hours 44 minutes for 11 expected reproductions as compared to its fuzzing duration of ≈15h. In contrast, OnePlus phone takes 2 hours 9 minutes for 13 reproductions as opposed to its fuzzing duration of ≈13h. Such difference in time comes from where bugs are located. Since most of the bugs for OnePlus are caused due to a single mutated packet (*rrcSetup*), few test cases generated by AIRBUGCATCHER can already reproduce majority of bugs for such target without additional test cases employing packet replay and flooding. In summary, AIRBUGCATCHER reproduces most crashes from the fuzzing log and takes significantly less time than that of the respective fuzzing campaign.

RQ2: How efficient is AIRBUGCATCHER to reproduce crashes? Figure 6 highlights the distribution of time taken to reproduce a bug by AIRBUGCATCHER. Specifically, Figure 6 captures the time period taken for AIRBUGCATCHER to automatically reproduce an expected bug, versus the number of bugs obtained within such time periods. We note that only the expected bugs are counted in Figure 6. Our findings reveal that most bugs can be successfully reproduced within four minutes for most devices, while only one or two crashes take more than 30 minutes for reproduction. More specifically, the time to automatically obtain the first expected bug for the five target devices (ESP32-WROOM-32, Cypress Board, OnePlus Phone, SIM8202G and ESP-WROVER-KIT) is 9.7, 0.8, 1.3, 12.9 and 56.4 minutes, respectively. Moreover, the average reproduction time for the five targets (i.e., total evaluation time divided by the number of expected reproductions) is 36.8, 21.6, 10, 28.8 and 86.8 minutes, respectively. These results highlight the efficiency and hence practicality of employing AIRBUGCATCHER into an existing fuzzing pipeline, which previously would require significant manual effort to reproduce bugs otherwise.

RQ3: How do the different design options impact the effectiveness of AIRBUGCATCHER?

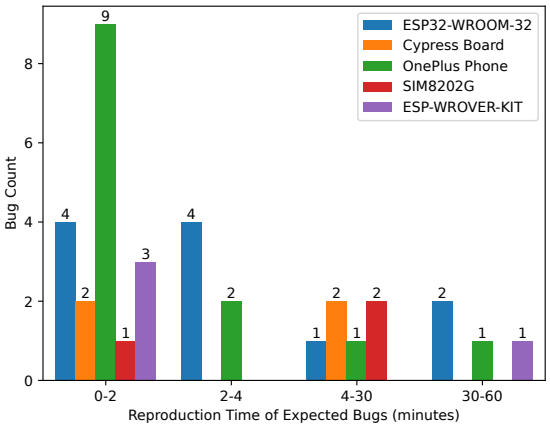

Figure 6: Distribution of # expected bugs w.r.t. time

To evaluate the effectiveness of AIRBUGCATCHER design, we conduct experiments related to different bug grouping and reproduction parameters. Firstly, to evaluate the design effectiveness of bug grouping as discussed in Section 3, we create fundamental modifications to AIRBUGCATCHER that allow us to enable or disable use of target logs in the Packet Analysis component (see Figure 3). This is to verify the effectiveness of target logs in the identification and reproduction of bugs. For AIRBUGCATCHER variant that does not analyze target logs, AIRBUGCATCHER falls back to use the *state information*, as discussed in Section 3. Secondly, we create three variants of AIRBUGCATCHER to evaluate the effectiveness of bug reproduction strategy (c.f., Test Case Generation in Section 2). Overall, we evaluate six variants of AIRBUGCATCHER (with or without support from targets logs) based on different types of bug grouping and reproduction:

1) **Mutation Only:** AIRBUGCATCHER only processes fuzzed packets that are mutated during Test Case Generation.
2) **Mutation + Replay:** Both mutated and replayed packets are included in Test Case Generation.
3) **All:** Adds flooding detection and generation to the Test Case Generation where applicable.

The results of our ablation study is shown in Table 5 and structured as follows: Firstly, For each evaluated target, the rows labeled ***with logs*** and ***w/o logs*** indicate variants of the Packet Analysis Component. Secondly, columns grouped

TABLE 5: Ablation Study of AIRBUGCATCHER

	Target	# Expected / Unexpected			# Unique Bugs	Max. Time
		Mutation only	Mutation + Replay	All		
w/o log	ESP32-WROOM-32	12 / 16	13 / 14	15 / 14	29	18 hr 40 min
with log		9 / 6	10 / 6	11 / 5	16	7 hr 54 min
w/o log	Cypress Board	3 / 1	5 / 0	5 / 0	6	2 hr 05 min
with log		N.A.	N.A.	N.A.	N.A.	N.A.
w/o log	OnePlus Phone	2 / 3	N.A.	N.A.	6	2 hr 49 min
with log		13 / 0	N.A.	N.A.	14	2 hr 09 min
w/o log	SIM8202G	3 / 1	N.A.	N.A.	4	1 hr 26 min
with log		N.A.	N.A.	N.A.	N.A.	N.A.
w/o log	ESP-WROVER-KIT	0 / 0	1 / 0	1 / 0	2	2 hr 01 min
with log		0 / 0	1 / 1	2 / 0	4	4 hr 02 min

into **Expected / Unexpected** show the reproduction of bugs using different AIRBUGCATCHER variations *(1)-(3)*, as discussed in the preceding paragraph. We note that only mutated packets are exhibited in the results for 5G devices (OnePlus Phone and SIM8202G) because there is no replayed packets present in their fuzzing logs. Moreover, some devices (Cypress Board and SIM8202G) do not produce target logs when a bug occurs. Thus, those devices are only experimented without support from targets logs. Lastly, column **Unique Bugs** shows the total number of grouped bugs obtained for each variation of the Packet Analysis component and the final column **Max. Time** represents the longest time to finish the evaluation of any respective variants of AIRBUGCATCHER.

We observe from Table 5 that the variation of AIRBUGCATCHER supporting logs may yield much less **Unique Bugs** than its counterpart without logs (for ESP32-WROOM-32). This is because, in complex stateful protocols (e.g., Bluetooth) and in the absence of target logs, AIRBUGCATCHER fails to group potentially similar bugs that are triggered in different states during the fuzzing campaign. This is particularly observed with target ESP32-WROOM-32, in which a total of 29 bugs are attempted to be reproduced by AIRBUGCATCHER. Consequently, the lack of target logs results in a worst-case time of 18 hours as opposed to only 7.54 hours when target logs are used. In contrast, for OnePlus phone and ESP-WROVER-KIT, the lack of logs contributed to lower identification of unique bugs (six and two) when compared to usage of logs (fourteen and four). This is because many different bugs were manifested in few 5G and Wi-Fi states only. Thus, using the state information for bug grouping resulted identical group for different bugs. We also observe that target logs provide more precise bug identifier to the Test Case Generation. This reduces the number of failures when reproducing bugs and time taken for bug reproduction.

Finally, Table 5 clearly indicates that the variant **All** generally reproduced more unique bugs as opposed to the other variants "Mutation Only" and "Mutation + Replay". This is particularly observed for ESP32-WROOM-32. This is because target ESP32-WROOM-32 contains multiple bugs that can only be triggered by replaying or flooding packets as opposed to only mutation.

Overall, the ablation results of AIRBUGCATCHER reveal that its proposed components are suitable and resilient to reproduce bugs over a range of different configurations.

TABLE 6: AIRBUGCATCHER Effectiveness w.r.t. Different Maximum Fuzzed Packet Generation (Max_{fpg}) and Maximum Trial Time (Max_{tt}).

Target	Max_{fpg}	Max_{tt}	# Expected	# Unexpected
ESP32-WROOM-32	3	10 min	9	5
		20 min	11	5
		40 min	11	5
	1	60 min	9	6
	2		11	5
	3		11	5
Cypress Board	3	10 min	3	0
		20 min	3	2
		40 min	5	0
	1	60 min	4	0
	2		5	0
	3		5	0
OnePlus Phone	3	10 min	12	1
		20 min	12	1
		40 min	13	0
	1	60 min	12	1
	2		12	1
	3		13	0
SIM8202G	3	10 min	1	3
		20 min	2	2
		40 min	3	1
	1	60 min	1	3
	2		3	1
	3		3	1
ESP-WROVER-KIT	3	10 min	1	1
		20 min	1	1
		40 min	1	1
	1	60 min	1	1
	2		1	1
	3		2	0

RQ4: How do the various execution parameters affect the effectiveness of AIRBUGCATCHER?

To evaluate the effectiveness of AIRBUGCATCHER running with different parameters, we carry out experiments using different values for Max_{fpg} and Max_{tt}. Table 6 presents the results of our experiment conducted on all five target devices. In our findings, a larger Max_{fpg} helps to reproduce more bugs. This is expected because certain bugs require more fuzzed packets to trigger than others. Furthermore, longer trial time contributes to the increase on the number of expected and unexpected bugs. One possible reason is that certain bugs are triggered by the fuzzed packets that are far away from the bug location. Consequently, these fuzzed packets are not generated during trials if Max_{tt} is small since fuzzed packets closer to the bug location are tried first. Finally, Max_{fpg} and Max_{tt} have little impact for the target OnePlus phone, because most of its bugs can be triggered by only one fuzzed packet. Therefore, a smaller window size and shorter maximum trial time suffice to trigger most bugs. Moreover, the impact of Max_{fpg} and Max_{tt} on ESP-WROVER-KIT is not obvious because its fuzzing log contains only four unique bugs and bugs are more difficult to reproduce in a complicated Wi-Fi protocol.

RQ5: How does AIRBUGCATCHER compare to existing tools?

TABLE 7: Effectiveness of Baseline (replay) Experiments

Target	# Reproduced Bugs in Trial / # Total Bugs in Fuzzing Log				
	# 1	# 2	# 3	# 4	# 5
ESP32-WROOM-32	13/190	16/190	23/190	13/190	16/190
Other Targets	0	0	0	0	0

In this experiment, we compare our AIRBUGCATCHER approach against baseline approaches used by other protocol fuzzers [3], [7].

To this end, we evaluate the intuitive approach of baseline fuzzers to reproduce bugs by replaying all TX packets from the fuzzing log (i.e. *Simple Replay*). The idea behind this approach is that a bug might be reproduced if an identical sequence of packets can be replayed to the target device, since the bug happens after such a sequence of packets were exchanged. However, this is often not a deterministic task, due to the fact that we only have control over TX packets but not RX packets which are received from the test device in real-time. Nonetheless, we run baseline experiment with the Simple Replay approach against all five target devices without grouping bugs. Then, for each target device, we attempt to generate and run as many PoCs as the number of crashes in the respective fuzzing logs. We also repeat the baseline experiments five times on each target device to potentially reduce the impact from non-deterministic behaviors of over-the-air protocols. During each trial, the control of packet transmission is performed by either *direct injection* of replayed TX packets (Bluetooth) or *replacing* (i.e., overwriting) TX packets of live communication with *replayed* TX packets in the sequence (5G, Wi-FI). The slight difference in the simple replay, based on target protocol, is due to the implementation details of the tool [8] used by AIRBUGCATCHER to communicate with target devices. Each PoC has a crash detection timeout of 60 seconds (the same as our other experiments). The result of our baseline experiments are presented in Table 7. We observe that simple replay approach is only able to trigger a few bugs (13-23 out of 190 across five trials) in the ESP32-WROOM-32 target, while no bugs were triggered at all in other targets. In contrast, AIRBUGCATCHER is able to reliably reproduce 16 unique bugs (out of 16 unique bugs in fuzzing log) for ESP32-WROOM-32 as highlighted in the Reproduced column of Table 4. The underwhelming performance of baseline is due to communication timeouts or the target expecting random packet field values (context) during bug reproduction. Since the target does not receive a response that corresponds to such field values, it drops the connection or send rejection messages during bug reproduction.

These results highlight the practicality of AIRBUGCATCHER to reproduce bugs under non-deterministic and adverse protocol communication scenarios.

6. Threats to Validity

Comprehensiveness of the Target Logs: The usage of target logs to identify unique bugs is one of the fundamental steps for precisely grouping bugs. If the target cannot provide logs, which contain crash dumps, the precision of the bug identifier is impaired. This may increase the time to reproduce bugs as can be observed in the results reported in Table 5. We mitigate this risk by enabling debug logs in the evaluated targets and by providing several mechanism of log collection such as via serial port, SSH and Logcat.

Number of bugs within bug groups: The number of bugs within a bug group may affect the capability of AIRBUGCATCHER to reproduce complex bugs. For example, if there are only few bugs within a group, then it reduces the variations of test cases to be tried by AIRBUGCATCHER (e.g., due to the lack of many locations for the bug). To mitigate this, we set the parameters Max_{lfi} and Max_{fpg} (see Table 1) appropriately such that many combinations of fuzzed packets are explored by AIRBUGCATCHER.

Completeness of Packet Filtering Rules: For AIRBUGCATCHER to create test cases, it needs to generate certain packet filters that match specific packets during the trial of the PoCs. Such is required for systematic transmission of mutated, replayed or flooding packets towards the target. In this context, it is important that the user provides reasonable *packet filtering rules* to the Packet Analysis component of AIRBUGCATCHER. We address this threat by creating packet filtering rules based on protocol standard, as also explored by prior works in protocol fuzzing [8].

Reproducible Target Builds: Since AIRBUGCATCHER relies on fuzzing logs of an existing fuzzing pipeline, using a slightly different target (or firmware version) within the same fuzzing pipeline may lead AIRBUGCATCHER to not reproduce bugs. This is due to deviations in target's behaviour. Such deviations might be caused by different responses from the target during PoCs trial or target logs mismatches (e.g., different crash dumps addresses) between what is received during fuzzing versus what is received during bug reproduction. We mitigate this by using the same target or firmware version throughout fuzzing and AIRBUGCATCHER test case generation to ensure reproduction of results.

7. Related Work

Bug Reproduction Within Protocol Fuzzers: Existing protocol fuzzers that offer reproduction of *stateful bugs* normally implement an approach that records the sequence of packets exchanged with the target [3], [7]. Subsequently, they replay benign or fuzzed packet sequences towards the target. While this approach may reproduce bugs for protocols that are mostly sequential and deterministic in nature, it does not consider deviations of the target's response due to non-deterministic behaviour of wireless protocols. Specifically for wireless communication, full control over the target's state during a test is not guaranteed. Therefore, replay techniques employed in prior works ought to fail reproduction of deeply rooted or ambiguous bugs. In contrast, AIRBUGCATCHER addresses this shortcoming by adopting a minimal number of state machine rules that can

guide its test case generation towards only specific parts of the communication with the target. Additionally, prior works attempt to reproduce bugs by sending the entire fuzzed sequence instead of communicating a minimal set of fuzzed packets that contribute to the bug. Consequently, AIRBUGCATCHER is more suitable to help triaging teams to focus on the relevant attack vector and hence fix the root cause of bugs faster.

Deterministic Network Replay or Instrumentation: Works that focus in reproducing [18], [20], [27] or monitoring [24] communication behaviour of network systems do not offer precise modification of protocol contents or interfacing with network protocols other than Ethernet. Aside from such works' usefulness in deterministically debugging wired network protocols, their support for construction of test cases is limited to only the recorded protocol packets. Moreover, collecting target logs is out-of-scope for such works. Consequently, analysis of target bugs are limited to hangs or performance degradation [18].

Wireless Sensor Network Replay: Tardis [25] and Minerva [23] offers packet replay facilities for wireless networks based on intrusive approaches. For example, Tardis requires access to the target's source code such that debugging code can be introduced into the target firmware. Subsequently, the new firmware generates instrumentation logs during normal operation of the target and is then used inside an emulator to replay packets offline. Similarly, Minerva enables replay of wireless packets through use of external debugging hardware (i.e., JTAG) attached to the target. In summary, both works require intrusive approaches which are not suitable to reproduce protocol bugs in closed wireless stacks such as the IoT devices targeted by AIRBUGCATCHER.

Reproduction of Bugs in Software: Our work is orthogonal to several parallel works that aim to reproduce the behaviour of both distributed software systems [19], [11], [26] and mobile applications [6], [29]. Concretely, these works are tailored to reproduce bugs within software binary that runs full-fledged operational system and hence they do not generalize to operate with external hardware, as often required with wireless fuzzers. In contrast, AIRBUGCATCHER focuses solely in the protocol interaction between two peers and hence it does not need to take into account the full behaviour of the underlying OS used by the targets during the fuzzing campaign.

Automated Exploit Generation: AIRBUGCATCHER runs orthogonally to prior works that automatically generate exploitable code based on open-source code or binary software. On the one hand, Siege [14] and Evomaster [2] are whitebox approaches that generate exploitable code based on static analysis of the target program. On the other hand, Crax [13] and Flowstitch [12] generate exploits solely using the program binary. While the latter works are well suitable to reproduce bugs in a blackbox target, these approaches are orthogonal to our objective, as we aim to reproduce bugs based on existing test case scenarios obtained from a wireless fuzzer. Moreover, leveraging such binary analysis [13], [12] may introduce significant technical challenges to reproduce protocol (e.g., stateful) bugs and to support static analysis on different architectures. In this context, AIRBUGCATCHER distinguishes itself by generating test cases while being inherently hardware and protocol agnostic. This is because AIRBUGCATCHER focuses on the bugs discovered by wireless fuzzers and integrates well to accelerate the triaging process.

Root Cause Diagnosis on Software Code: There are several works that can indicate the root cause of bugs within software code via static or dynamic analysis [17], [4], delta debugging [28], as well as pinpointing the commits in which bugs were firstly introduced [1]. However, such works are neither directly applicable to stateful protocols fuzzing, nor applicable to closed-source IoT targets.

8. Conclusion

In this paper, we propose and implement AIRBUGCATCHER, to automatically and systematically reproduce bugs in wireless IoT devices to accelerate the troubleshooting, triaging and fixing process of vulnerable IoT devices. We show that the non-deterministic nature of wireless protocols demand a fundamentally different approach for bug reproduction, as simple replay-based techniques fail to preserve the dynamic context during wireless communication. Moreover, our AIRBUGCATCHER approach provides a range of offline analysis to reduce the size of PoC to only a few packets, which, we believe should significantly help in understanding the root cause of bugs for the vendors. Apart from reliably reproducing wireless implementation bugs, we believe the capabilities embodied in AIRBUGCATCHER can be leveraged for several other future research directions in wireless security. For example, the extracted filtering conditions by AIRBUGCATCHER are used for test-case generation, however, such conditions may also assist in over-the-patch creation or input repair to protect the vulnerable IoT devices. Moreover, this input-repair process may also guide the fuzzing process to discover other vulnerabilities that do not appear in the fuzzing log. We hope AIRBUGCATCHER provides a valuable tool to improve the security testing pipeline of IoT devices. To advance research in the area of wireless security and testing, we have made our tool and all experimental data available in the following:

https://github.com/asset-group/air-bug-catcher

Acknowledgement: This research is partially supported by MOE Tier 2 grant (Award number MOE-T2EP20122-0015), National Research Foundation, Singapore and Infocomm Media Development Authority under its Future Communications Research & Development Program (Award number FCP-SUTD-RG-2022-017) and National Research Foundation, Singapore, under its National Satellite of Excellence Programme "Design Science and Technology for Secure Critical Infrastructure: Phase II" (Award No: NRF-NCR25-NSOE05-0001). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the respective funding agencies.

References

[1] Rui Abreu, Franjo Ivančić, Filip Nikšić, Hadi Ravanbakhsh, and Ramesh Viswanathan. Reducing time-to-fix for fuzzer bugs. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1126–1130, 2021.

[2] Andrea Arcuri. Restful api automated test case generation with evomaster. *ACM Trans. Softw. Eng. Methodol.*, 28(1), jan 2019.

[3] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3255–3272, Boston, MA, August 2022. USENIX Association.

[4] Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Adithya Abraham Philip. Orca: Differential bug localization in Large-Scale services. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 493–509, Carlsbad, CA, October 2018. USENIX Association.

[5] Hongjian Cao, Lin Huang, Shuwei Hu, Shangcheng Shi, and Yujia Liu. Owfuzz: Discovering wi-fi flaws in modern devices through over-the-air fuzzing. In *WISEC*, pages 263–273. ACM, 2023.

[6] Sidong Feng and Chunyang Chen. Prompting is all you need: Automated android bug replay with large language models. In *ICSE*, pages 67:1–67:13. ACM, 2024.

[7] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 337–350, New York, NY, USA, 2021. Association for Computing Machinery.

[8] Matheus E. Garbelini, Vaibhav Bedi, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. BrakTooth: Causing havoc on bluetooth link manager via directed fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1025–1042, Boston, MA, August 2022. USENIX Association.

[9] Matheus E. Garbelini, Zewen Shang, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. Towards automated fuzzing of 4g/5g protocol implementations over the air. In *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, pages 86–92, 2022.

[10] Matheus E. Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sun Sumei, and Ernest Kurniawan. SweynTooth: Unleashing mayhem over bluetooth low energy. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 911–925. USENIX Association, July 2020.

[11] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: replication at the speed of multi-core. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[12] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of Data-Oriented exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 177–192, Washington, D.C., August 2015. USENIX Association.

[13] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Chung-Wei Lai, Han-Lin Lu, and Wai-Meng Leong. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 78–87, 2012.

[14] Emanuele Iannone, Dario Di Nucci, Antonino Sabetta, and Andrea De Lucia. Toward automated exploit generation for known vulnerabilities in open-source libraries. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 396–400, 2021.

[15] I. Karim, A. Ishtiaq, S. Hussain, and E. Bertino. Blediff: Scalable and property-agnostic noncompliance checking for ble implementations. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 3209–3227, 2023.

[16] Hongil Kim, Jiho Lee, Eunkyu Lee, and Yongdae Kim. Touching the untouchables: Dynamic security analysis of the LTE control plane. In *IEEE Symposium on Security and Privacy*, pages 1153–1168. IEEE, 2019.

[17] Ao Li, Shan Lu, Suman Nath, Rohan Padhye, and Vyas Sekar. ExChain: Exception dependency analysis for root cause diagnosis. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 2047–2062, Santa Clara, CA, April 2024. USENIX Association.

[18] Yuliang Li, Rui Miao, Mohammad Alizadeh, and Minlan Yu. DETER: Deterministic TCP replay for performance diagnosis. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 437–452, Boston, MA, February 2019. USENIX Association.

[19] Christopher Lidbury and Alastair F. Donaldson. Sparse record and replay with controlled scheduling. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 576–593, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 417–429, Santa Clara, CA, July 2015. USENIX Association.

[21] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets. In *USENIX Security Symposium*, pages 19–36. USENIX Association, 2020.

[22] Zewen Shang, Matheus E. Garbelini, and Sudipta Chattopadhyay. U-fuzz: Stateful fuzzing of iot protocols on cots devices. *17th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2024.

[23] Philipp Sommer and Branislav Kusy. Minerva: distributed tracing and debugging in wireless sensor networks. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, New York, NY, USA, 2013. Association for Computing Machinery.

[24] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with SwitchPointer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 453–456, Renton, WA, April 2018. USENIX Association.

[25] Matthew Tancreti, Vinaitheerthan Sundaram, Saurabh Bagchi, and Patrick Eugster. Tardis: software-only system-level record and replay in wireless sensor networks. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, IPSN '15, page 286–297, New York, NY, USA, 2015. Association for Computing Machinery.

[26] Wei Wang, Zhiyu Hao, and Lei Cui. Clusterrr: a record and replay framework for virtual machine cluster. In *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2022, page 31–44, New York, NY, USA, 2022. Association for Computing Machinery.

[27] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. OFRewind: Enabling record and replay troubleshooting for networks. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, Portland, OR, June 2011. USENIX Association.

[28] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[29] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G.J. Halfond. Recdroid: Automatically reproducing android application crashes from bug reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 128–139, 2019.

Appendix

Packet Filtering Rules

In this section, we provide packet filtering rules for the OTA protocols used in our experiment. Precisely, Figure 7, Figure 8 and Figure 9 demonstrate the packet filtering rules for Bluetooth Classic, 5G NR and Wi-Fi respectively.


```
"Mapping": [
    {
        "Filter": "btsdp",
        "StateNameField": "btsdp.pdu"
    },
    {
        "Filter": "bta2dp",
        "StateNameField": "bta2dp.codec"
    },
    {
        "Filter": "btavrcp",
        "StateNameField": "btavrcp.notification.event_id"
    },
    {
        "Filter": "btavdtp",
        "StateNameField": "btavdtp.signal_id"
    },
    {
        "Filter": "btrfcomm",
        "StateNameField": "btrfcomm.frame_type"
    },
    {
        "Filter": "btl2cap",
        "StateNameField": "btl2cap.cmd_code"
    },
    {
        "Filter": "btbrlmp.op == 3",
        "StateNameField": "btbrlmp.opinre"
    },
    {
        "Filter": "esp32_btlmp",
        "StateNameField": [
            "btbrlmp.eop",
            "btbrlmp.op"
        ]
    },
    {
        "Filter": "fhs",
        "StateNameField": "btbbd.type"
    }
]
```


Figure 7: Bluetooth Classic Packet Filtering Rules


```
"Mapping": [
    {
        "Filter": "nas-5gs",
        "StateNameField": [
            "nas_5gs.sm.message_type",
            "nas_5gs.mm.message_type"
        ]
    },
    {
        "Filter": "nr-rrc",
        "StateNameField": [
            "nr-rrc.c1"
        ]
    },
    {
        "Filter": "rlc-nr",
        "StateNameField": [
            "rlc-nr.am.cpt",
            "rlc-nr.am.dc"
        ]
    },
    {
        "Filter": "mac-nr",
        "StateNameField": [
            "mac-nr.ulsch.lcid",
            "mac-nr.dlsch.lcid",
            "mac-nr.rnti-type"
        ]
    }
]
```


Figure 8: 5G NR Packet Filtering Rules


```
"Mapping": [
    {
        "Filter": "tls",
        "StateNameField": [
            "record.content_type"
        ]
    },
    {
        "Filter": "eap",
        "StateNameField": [
            "eap.type",
            "eap.code"
        ]
    },
    {
        "Filter": "eapol",
        "StateNameField": [
            "eapol.keydes.type",
            "eapol.type"
        ]
    },
    {
        "Filter": "wlan.fixed.action_code",
        "StateNameField": "wlan.fixed.action_code"
    },
    {
        "Filter": "wlan",
        "StateNameField": "wlan.fc.type_subtype"
    }
]
```


Figure 9: Wi-Fi Packet Filtering Rules