USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

LLFuzz: An Over-the-Air Dynamic Testing Framework for Cellular Baseband Lower Layers

Tuan Dinh Hoang and Taekkyung Oh, *KAIST;* CheolJun Park, *Kyung Hee University;* Insu Yun and Yongdae Kim, *KAIST*

https://www.usenix.org/conference/usenixsecurity25/presentation/hoang

This paper is included in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

LLFuzz: An Over-the-Air Dynamic Testing Framework for Cellular Baseband Lower Layers

Tuan Dinh Hoang[1], Taekkyung Oh[1], CheolJun Park[2*], Insu Yun[1*], Yongdae Kim[1]

[1]*KAIST,* [2]*Kyung Hee University*

Abstract

Memory corruptions in cellular basebands are critical because they can be remotely exploited over-the-air, resulting in severe consequences such as remote code execution, denial of service, and information leakage. While previous research has made significant contributions to detecting memory corruptions in basebands, particularly in layer 3 protocols (*e.g.*, NAS and RRC), the lower layers have received comparatively less attention, with only a few works exploring them in a limited and non-systematic manner.

In this paper, we present Lower-Layer Fuzzer (LLFuzz), a novel over-the-air dynamic testing framework that discovers memory corruptions in baseband lower layers. LLFuzz systematically targets lower layers, which are the PDCP, RLC, MAC, and PHY layers of the cellular stack. Testing these layers presents unique challenges due to their multiple channels and packet structures that can be dynamically configurable. To address these complexities, LLFuzz implements a channel-driven, configuration-aware fuzzing approach to systematically explore multiple channels. During the testing process, LLFuzz actively modifies layer-specific configurations through signaling messages to trigger and test diverse packet structures, particularly those rarely used in commercial networks. Moreover, LLFuzz leverages 3GPP specifications to generate test cases tailored to the packet structures of the lower layers. This ensures that the test cases are syntactically valid and capable of reaching the target layers without being prematurely discarded. In our evaluation of 15 commercial basebands from five major vendors, LLFuzz uncovered nine previously unknown memory corruptions: two in PDCP, two in RLC, and five in MAC layers. These findings demonstrate LLFuzz's effectiveness in finding critical memory corruptions in baseband lower layers.

1 Introduction

Cellular basebands play a crucial role in mobile communication systems, handling a wide range of tasks—from managing critical signaling protocols to providing essential services such as voice calls and data transmission. Unfortunately, their critical functions, coupled with the inherently open nature of wireless interfaces, make them attractive targets for attackers capable of transmitting over-the-air signals via Software Defined Radios (SDRs). Consequently, various attack models have been developed to exploit baseband vulnerabilities, including passive eavesdropping [17, 25, 38], fake base stations [36], signal overshadowing [16, 37, 60], and man-in-the-middle attacks [50]. These attacks utilize numerous vulnerabilities across multiple baseband layers, posing substantial threats to user security and privacy.

One notable class of vulnerabilities in basebands is memory corruption. This vulnerability occurs due to programming errors, such as mishandling pointers when decoding messages from cell towers. The consequences of memory corruption can be severe, leading to arbitrary code execution, denial of service, and information leakage. For that reason, memory corruption has become a topic of great interest to both academia and industry. For example, Marco Grassi *et al.* exploited a vulnerability in the Huawei Kirin baseband to modify the International Mobile Equipment Identity (IMEI) number [23]. Google Project Zero also reported several memory bugs in Samsung's Exynos basebands [61], and more recently, Berard

Table 1: Supported layers of existing tools and LLFuzz.

Tool	Gen	Approach	Bug	Layers	ST	CH	CO
LTEFuzz [31]	LTE	OTA	L	L3	–	–	–
DoLTEst [43]	LTE	OTA	L	L3	✓	–	–
BaseComp [29]	LTE	Reversing	L	L3	–	–	–
BaseSpec [30]	LTE	Reversing	M	L3	–	–	–
BaseSAFE [39]	LTE	Emulation	M	L3	–	–	–
FirmWire [24]	LTE	Emulation	M	L3	–	–	–
BaseBridge [32]	LTE	Emulation	M	L3	✓	–	–
LORIS [48]	LTE, 5G	Emulation	M	L3	✓	–	–
Goos *et al.* [15]	2.5G	Emulation	M	L2, L3	–	–	–
5Ghoul [20]	5G	OTA	M	L2, L3	✓	–	–
LLFuzz	LTE	OTA	M	L1, L2	✓	✓	✓

ST: Stateful Testing, **CH**: Channel-driven Testing, **CO**: Configuration-aware Testing, **L**: Logical Bug, **M**: Memory Bug, **OTA**: Over-The-Air Testing.

* They are co-corresponding authors.

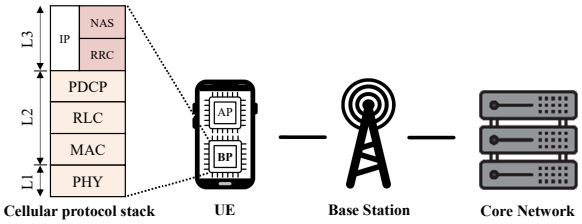


Figure 1: Cellular network architecture and stack.

and Dehors demonstrated a zero-click RCE on Tesla through its cellular modem [10].

Given the critical impact of memory corruptions, researchers have proposed several techniques to detect them in basebands. Many memory corruption vulnerabilities have been revealed through reverse engineering [12, 13, 21, 22, 23, 29, 30, 33, 57, 58], emulation-based fuzzing [24, 32, 39, 48], and over-the-air fuzzing [41, 47, 56] approaches. While these works have been effective, they primarily focused on layer 3 (*i.e.*, NAS and RRC), as these are the most well-known control plane protocols in cellular networks.

Layer 2 (*i.e.*, MAC, RLC, PDCP) and layer 1 (*i.e.*, PHY) are often overlooked, despite their critical role in basebands. A few works (Table 1) have attempted to fuzz layer 2, but they either targeted older cellular generations (*i.e.*, 2G, 3G) [15] or did not systematically test the lower layers [19, 20]. For instance, Goos and Muench [15] extended FirmWire emulation to support fuzzing layer 2, but it only covered the lower layers of General Packet Radio Service (GPRS) and Global System for Mobile Communications (GSM), which are older cellular generations. Garbelini *et al.* proposed a tool called 5Ghoul [20] that can fuzz layer 2 and layer 3 of the 5G basebands. They identified three layer 2 vulnerabilities by mutating elementary protocol messages implemented in open-source cellular networks, guided by grammar information extracted from Wireshark's packet dissector. However, their approach cannot effectively cover rarely used messages and configuration-dependent behavior of basebands, which are often critical for uncovering corner-case vulnerabilities.

In this paper, we present LLFuzz, an over-the-air dynamic testing framework designed to discover memory corruptions in the lower layers of cellular basebands. Testing these layers is challenging due to the presence of multiple channels and the fact that various packet structures are configurable by the upper layer (*i.e.*, the RRC layer). In particular, during the Attach procedure, the cell tower transmits signaling messages (*i.e.*, RRC messages) to the baseband, which activate lower-layer channels (*i.e.*, logical channels) and provide layer-specific configurations that define the corresponding packet structures. Without accounting for these properties, it is not possible to systematically test diverse packet structures in the lower layers.

To address these challenges, LLFuzz employs a channel-driven, configuration-aware fuzzing approach for in-depth testing of the lower layers. LLFuzz first tracks the establishment of logical channels using our newly defined channel-oriented states, and then actively modifies layer-specific configurations to test diverse packet structures. This approach allows LLFuzz to explore a broader range of packet structures, particularly those rarely used by commercial cell towers or not supported by open-source tools. Given the slow speed of over-the-air testing, LLFuzz also relies on specifications when generating test cases. As a result, LLFuzz's test cases are less likely to be prematurely rejected by basebands, increasing the likelihood of uncovering memory corruptions.

We evaluated LLFuzz on 15 commercial basebands from five major vendors: Qualcomm, MediaTek, Samsung, Google Tensor, and Huawei Kirin. Our evaluation uncovered a total of nine memory corruptions across multiple layers: five in the MAC layer, two in the RLC layer, and two in the PDCP layer. These findings demonstrate that memory corruptions are prevalent in the lower layers of basebands, emphasizing the need to address vulnerabilities in these layers with the same level of attention as those in the higher layers.

In summary, this paper makes the following contributions:

- We present LLFuzz, an over-the-air dynamic testing framework specifically designed to detect memory corruptions in the lower layers of cellular basebands.
- We propose a channel-driven, configuration-aware fuzzing approach that enables systematic testing of lower-layer protocols.
- We evaluated LLFuzz on 15 commercial basebands from five major vendors and found a total of nine memory corruptions.
- To foster future research, we release LLFuzz as an open-source tool: https://github.com/SysSec-KAIST/LLFuzz.

2 Background

2.1 Cellular Protocol Stack

Figure 1 illustrates the overall cellular network architecture and protocol stack. The network consists of three main components: the User Equipment (UE), the base station (eNB), and the core network. There are two main processors in the UE, the application processor (AP) and the baseband processor (BP). The BP consists of three main layers: 1) the physical layer (layer 1 - PHY), 2) the data link layer (layer 2), and 3) the network layer (layer 3). Layer 2 is further divided into three sub-layers: the Medium Access Control (MAC) sub-layer, the Radio Link Control (RLC) sub-layer, and the Packet Data Convergence Protocol (PDCP) sub-layer. Hereafter, these sub-layers will be referred to as layers for simplicity.

2.2 Lower Layers in Basebands

Lower layers (*i.e.*, layer 1 and layer 2) in the cellular protocol stack facilitate radio communication between UE and eNB by handling critical functions: decoding signals, demultiplexing

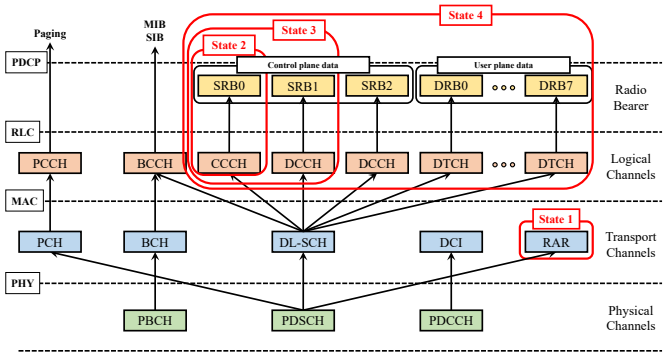


Figure 2: LTE logical channels and state definitions [51].

data streams, mapping data to logical channels, and transferring it to upper layers (*i.e.*, layer 3). These processes ensure seamless interaction between network components, forming the foundation for reliable cellular communication.

As cellular standards have evolved, lower-layer structures have grown more complex, incorporating advanced mechanisms like Multiple-Input Multiple-Output (MIMO), Carrier Aggregation (CA), Timing Advance, and power control in LTE and 5G—reflecting demands for higher throughput and improved performance.

PHY layer. The PHY layer handles all the physical aspects of the radio interface, including modulation, coding, and transmission of data over the air interface. To guide UE in decoding the incoming downlink signals and sending uplink signals, the eNB sends the Downlink Control Indicator (DCI) messages, which contain the Radio Network Temporary Identifier (RNTI) of the target UE. This DCI acts as the header of the data packets sent in the PHY layer. It consists of time-domain and frequency-domain resource allocation information, modulation and coding schemes, and other control information.

There are multiple formats of DCIs, each used for different transmission modes (TMs). For example, the DCI format 1 is used in TM1, in which UE and eNB only use a single antenna. Figure 3d shows the structure of DCI format 1 in a base station with a 20MHz channel, Frequency Division Duplexing (FDD), allocation type 0, and no CA. This DCI 1 consists of a Resource allocation header (`R`), Resource Block (`RB`) assignment type 0, Modulation and Coding Scheme (`MCS`), Hybrid Automatic Repeat reQuest (HARQ) process (`H`), Redundancy Version (`RV`), and Power Control (`PC`) fields.

MAC layer. The MAC layer is responsible for managing random access and multiplexing data streams from multiple channels. Its packets, referred to as MAC Protocol Data Units (PDUs), include several types. For example, the Downlink Shared Channel (DL-SCH) PDU delivers user data and control information, while the Random Access Response (RAR) PDU allocates radio resources and identifiers to UEs during the Random Access Channel (RACH) procedure.

Both DL-SCH and RAR PDUs can contain multiple sub-headers and sub-payloads, with the sub-payloads referred to as MAC Service Data Units (SDUs). These PDUs support various sub-header formats, each with multiple fields. For example, the DL-SCH sub-header includes fields like Logical Channel ID (`LCID`), Length (`L`), and Extension (`E`), as shown in Figure 3a. The `LCID` identifies the logical channel, the Length specifies the size of the MAC SDU, and the Extension indicates the presence of additional sub-headers. Additionally, DL-SCH PDUs can carry MAC Control Elements (CEs) to manage the radio connection between the UE and the eNB.

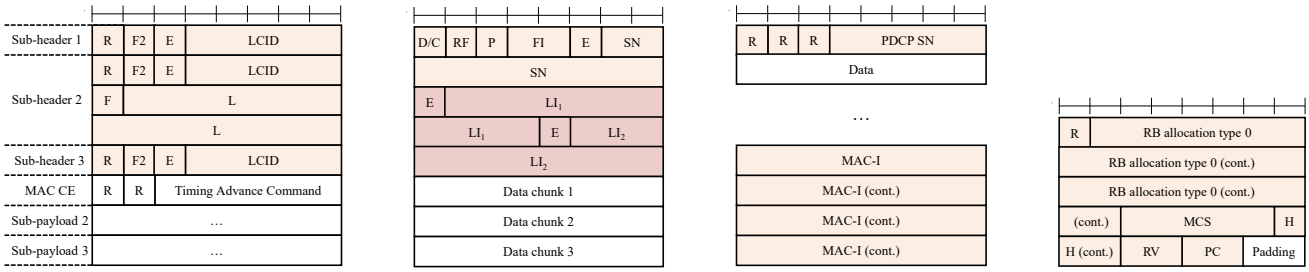
RLC layer. The RLC layer is responsible for the segmentation and reassembly of data packets, reordering data sequences, and performing layer 2 retransmissions. Data transmission in this layer can be configured in three modes: Transparent Mode (TM), Unacknowledged Mode (UM), and Acknowledged Mode (AM); each mode has its own packet structures. Figure 3b shows an example of the RLC AM Data PDU structure. This structure consists of a fixed header, an extension header, and a payload. Some important fields in the fixed header are Sequence Number (`SN`), Polling Bit (`P`), Extension Bit (`E`), and Length Indicator (`LI`). Notably, when the RLC PDU contains multiple segments, the extension includes multiple [`E`, `LI`] pairs to indicate the length of each segment.

PDCP layer. The PDCP layer ensures security and integrity of transmitted data. It compresses IP headers (when enabled), applying ciphering and integrity protection, and adding a PDCP header that varies between user data and signaling messages. One example of the PDCP Data PDU structure for signaling messages is shown in Figure 3c. This structure consists of a header with a sequence number (`SN`) field, payload, and Message Authentication Code (`MAC-I`), though the `MAC-I` is only present in control plane messages where integrity protection is required. Note that the PDCP layer only encrypts and protects the payload (*i.e.*, layer 3 data), while the PDCP header remains unprotected even after the Authentication and Key Agreement (AKA) procedure.

Logical channels in lower layers. In LTE, logical channels define data types transmitted between UE and eNB. To support this, the MAC layer uses the LCID field in its header to identify the target channel. Also, the RLC and PDCP layers are divided into multiple entities, each handling data from a specific logical channel. There are 5 main logical channels in LTE: 1) Broadcast Control Channel (BCCH), 2) Paging Control Channel (PCCH), 3) Common Control Channel (CCCH), 4) Dedicated Control Channel (DCCH), and 5) Dedicated Traffic Channel (DTCH) (see Figure 2). Since there are two different DCCH channels (for signaling before and after AKA), we refer to them as DCCH-1 and DCCH-2, respectively.

2.3 Attach Procedure

The UE performs the Attach procedure to connect to the network (Figure 4). During this process, it establishes a radio connection with the eNB and completes the AKA procedure to secure the connection. From a lower-layer perspective, this procedure establishes logical channels for transmitting



(a) MAC DL-SCH PDU structure. **(b)** RLC AM Data PDU structure. **(c)** PDCP Data PDU structure. **(d)** DCI structure.

Figure 3: Examples of structures in lower layers.

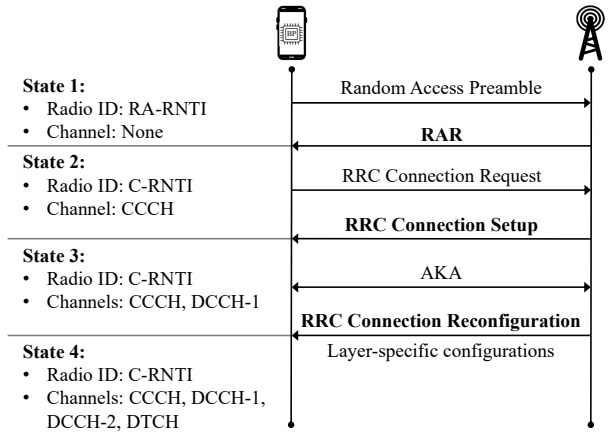


Figure 4: LTE Attach procedure and channel-oriented states for lower layers.

different types of data. Specifically, the RAR message enables CCCH, RRC Connection Setup enables DCCH-1, and RRC Connection Reconfiguration enables DCCH-2 and DTCH.

Layer-specific configurations. RLC and PDCP packets mapped to DTCH can be customized with various configurations delivered via RRC Reconfiguration messages during the Attach procedure. For example, the RLC UM Data PDU can be configured with either a 5-bit or 10-bit `SN`, while the AM Data PDU supports 11/15-bit `LI` and 10/16-bit `SN` fields.

3 Gaps in Previous Works

There have been several approaches to detect memory corruption vulnerabilities in cellular basebands. However, previous works have the following gaps, which motivate our work.

Mainly focused on layer 3 protocols. Most prior work primarily targets memory corruption bugs in layer 3 protocols (*e.g.*, NAS and RRC). BaseSpec [30] and BaseComp [29] performed reverse engineering and conducted a comparative analysis between layer 3 implementations and their corresponding cellular specifications. Although effective, these approaches are constrained to layer 3 protocol messages that are explicitly defined in tabular formats. To improve automation and scalability, subsequent studies such as FirmWire [24] and BaseSAFE [39] introduced emulation-based approaches, enabling direct exposure of fuzzing entry points for layer 3 functionality. Complementary to emulation-based efforts, over-the-air fuzzing techniques—including those by Mulliner *et al.* [41], Wang *et al.* [56], and Potnuru and Nakarmi [47]—have targeted layer 3 protocols and services through the mutation of live wireless traffic. Despite their different methodologies, both emulation-based and over-the-air approaches have mainly focused on testing layer 3, overlooking the lower layers of the baseband.

Layer 3 has attracted more attention than the lower layers due to its broader protocol definition, which supports critical functions like authentication and mobility management (Table 1). However, as cellular technologies have evolved from LTE to LTE-A and LTE-Pro, the lower layers have become increasingly complex, incorporating numerous new features and dynamic configurations. Moreover, the lower-layer messages are never protected by encryption or integrity checks, leaving them exposed to adversarial manipulation. Therefore, memory corruption vulnerabilities in the lower layers can be just as critical as those in layer 3, yet remain underexplored—a gap this work aims to address.

Limited support for lower layers. A few prior efforts have explored testing at layer 2 of cellular basebands [15, 19, 20], but their scope and applicability remain limited. Goos and Muench [15] extended FirmWire [24] to emulate GSM and GPRS layer 2 protocols, and applied AFL++ [18] to detect memory corruptions in Samsung basebands. While their work demonstrates the feasibility of fuzzing at layer 2, it is limited to earlier-generation protocols. FirmWire currently lacks support for LTE lower-layer entry points, and extending it would require significant reverse engineering. Also, Qualcomm basebands rely on the proprietary Hexagon architecture, making emulation challenging. Meanwhile, 5Ghoul [20] applied over-the-air fuzzing to layers 2 and 3 of 5G basebands. By mutating legitimate packets captured from an open-source cellular network using Wireshark's packet dissector, they uncovered three layer 2 bugs. However, this approach is limited to a default gNB configuration and does not account for the wide range of packet formats induced by layer-specific signaling (*e.g.* triggering short SN fields in RLC UM). Also, since 5Ghoul mutates default packets from the open-source gNB, the generated test cases exhibit limited structural diversity and fail

to cover malformed inputs such as very short packets or abnormally many sub-headers. Moreover, random mutations often corrupt critical fields (*e.g.* logical channel IDs), causing early rejection before reaching the intended processing layer. Overall, for lower layers, existing methods either lack support for recent cellular generations or rely on oversimplified fuzzing configurations, and therefore do not offer the depth and flexibility required to systematically uncover bugs in modern implementations of these layers.

4 Overview

4.1 Scope of LLFuzz

Our ultimate goal is to develop a systematic approach to detect memory corruption vulnerabilities in the lower layers of cellular basebands, as these layers have not been adequately covered in previous works. In particular, we aim to test the decoding functions in layer 2 (*i.e.*, MAC, RLC, and PDCP) and layer 1 (*i.e.*, PHY) of the LTE protocol stack. Instead, we do not consider the upper layers such as NAS and RRC. Moreover, our work is limited to memory corruption bugs that can be identified using a universal bug oracle (*i.e.*, crash logs). However, our test case generation method is generic, so it can be repurposed to detect other bugs when it is combined with a special bug oracle like LTEFuzz [31].

4.2 Threat Model

We consider an attacker who aims to exploit memory corruption vulnerabilities in basebands through the over-the-air interface. Typically, this attacker is equipped with a software-defined radio (SDR) and share the same cell as the target baseband. Then, the attacker can send arbitrary packets to the target baseband using Fake Base Station (FBS) [36], Signal overshadowing (SigOver) [60], or Man-in-the Middle (MitM) [50] attacks. Similar to previous works [19, 20, 41, 47, 56], we assume the attacker lacks knowledge of the baseband's cryptographic context and therefore cannot compromise the encryption or the integrity of legitimate packets. We also assume that the attacker knows the target's RNTI, which can be inferred from various existing attacks [27, 38, 42]. Even without this knowledge, the attacker can conduct blind attacks by observing all active RNTIs within the cell [17, 25, 38].

4.3 Challenges and Approaches

In this section, we discuss technical challenges and our approaches to address them.

4.3.1 Complex Messages with OTA Testing

Unlike emulation-based testing [24, 39], over-the-air testing [20, 31, 43] can be applied to basebands from various vendors, but it comes with two significant limitations. First, this testing process for basebands is extremely slow. While existing fuzzers like libFuzzer [1] or AFL++ [18] can perform thousands of tests per second, over-the-air testing can approximately take 10 seconds per test case, as described in [43]. Second, as black box testing, coverage-guided fuzzing cannot be applied, which requires monitoring the program's behavior. Consequently, random data generation approaches used by many fuzzers like 5Ghoul [20] and AFL++ [18] fail to adequately test the lower layers with complex constraints. While DoLTEst [43] and LTEFuzz [31] suggested specialized test case generation to address this issue, their solutions are limited to the upper layers.

Our approach: Specification-guided test case generation. LLFuzz employs a specification-guided approach to generate test cases for various packet structures in the lower layers. This approach is similar to existing works [28, 31, 43]; however, we define new test case generation rules for the lower layers, as their grammars are fundamentally different from those of the upper layers. In particular, we carefully review the specifications to identify possible packet structures and their constraints (*i.e.*, field sizes in bits). Based on these structures and constraints, LLFuzz determines appropriate values for field mutations, ensuring that changes do not affect adjacent fields. It also considers the relevant states and configurations. This approach allows LLFuzz to generate test cases that are not prematurely rejected by basebands, yet are still syntactically incorrect enough to trigger bugs.

4.3.2 Diverse Messages across Multiple Channels

During our analysis of the lower layers, we discovered that, unlike layer 3, lower layers handle multiple channels simultaneously. In fact, layer 3 also has multiple channels, but most prior works [31, 43] focus on the control channel, which is the most interesting for security testing. On the other hand, the lower layers contain various logical channels, each of which processes different types of messages. Thus, in-depth exploration of these layers requires careful consideration of the relevant logical channels.

Unfortunately, the current specifications for the lower layers do not explicitly provide UE states based on logical channels. Instead, they only describe architectures, packet structures for each protocol, and implementation guidelines for various procedures. For instance, the MAC layer specification [4] implicitly defines two states—*Random Access Procedure* and *RRC Connected*. Unfortunately, these implicit states are insufficient to cover all the states and transitions in the MAC layer. More seriously, the specification does not provide clear definitions of states or transitions for other lower layers such as PHY, RLC, and PDCP. If we do not consider these states, the baseband will discard our test cases that are irrelevant to the current channel status, resulting in inefficient testing.

Our approach: Channel-driven stateful testing. To address this, LLFuzz performs channel-driven, stateful testing. To

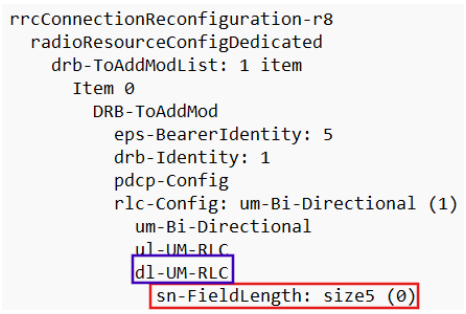


Figure 5: RRC Connection Reconfiguration message configuring the RLC UM Data PDU with a 5-bit Sequence Number.

this end, we newly define four channel-oriented states for the lower layers based on the establishment of logical channels (Figure 4). Our definition covers UE states and transitions during the Attach procedure, which are not explicitly defined in the specifications. Using these states, LLFuzz determines which logical channels are established, which structures to use, and which mappings to apply when generating and sending test cases. This enables LLFuzz to systematically explore all logical channels for the lower-layer testing.

4.3.3 Configurable Packet Structures during Attach Procedure

While the lower layers include multiple logical channels, the packet structures of certain channels (*e.g.*, DTCH in RLC, DRB (Data Radio Bearer) in PDCP) can be further customized through various layer-specific configurations during the Attach procedure (Table 2). As a result, effective testing of these logical channels requires not only generating test cases for each channel but also considering all potential configurations. Unfortunately, commercial and open-source base stations [26, 53] typically utilize only a subset of these configurations, limiting the diversity of packet structures observed in their logs. Consequently, it is inefficient to solely mutate these packets for generating test cases (*e.g.*, 5Ghoul [20] or LTE-Fuzz [31]), as it may fail to trigger and explore all possible configurations.

Our approach: Configuration-aware testing. To address this, LLFuzz leverages a configuration-aware testing approach. As mentioned in §2, layer-specific configurations can be delivered to the UE through RRC Connection Reconfiguration messages (Figure 5). LLFuzz utilizes this message to set the UE with the target configuration, and then generates and transmits test cases accordingly. This approach enables LLFuzz to trigger and test diverse layer-specific configurations across channels, which were insufficiently addressed by previous approaches. For instance, the DTCH in the RLC layer can operate in UM mode, which can be configured to use either a 5-bit or 10-bit SN. By default, most commercial and open-source eNBs only leverage a 10-bit SN. LLFuzz can specifically test the 5-bit SN by sending an RRC Reconfiguration message with the `sn-FieldLength` set to 5.

5 Design

5.1 LLFuzz overview

As shown in Figure 6, our system design comprises three main processes: 1) specification analysis, 2) over-the-air testing, and 3) post analysis. First, we manually analyze 3GPP specifications to define the channel-oriented states (§5.2) and identify all packet structures for lower layers (§5.3). Based on this, LLFuzz generates test cases (§5.4) and injects them over-the-air to the target baseband (§5.5). During this phase, LLFuzz detects baseband crashes (§5.6). Finally, we conduct a manual analysis of bug candidates and identify their root causes using a vendor-specific debug mode (§5.7).

5.2 Channel-Oriented States in Lower Layers

We define channel-oriented states for lower layers by manually analyzing 3GPP specifications and mapping the relationships between logical channels and UE states during the Attach procedure. Since 3GPP specifications do not explicitly define lower-layer states or the transitions of logical channels (Figure 2), this mapping provides a crucial foundation for test case generation.

Among the six logical channels described in §2.2, the CCCH, DCCH-1, DCCH-2, and DTCH channels are involved in the Attach procedure. Each of these channels plays a key role in carrying different types of signaling and user-plane messages and has different packet structures, detailed in §5.3. Since these channels are progressively established during the Attach procedure, we define four channel-oriented states based on active channels, as shown in Figure 4. The states are as follows:

State 1. This state is initiated when the UE transmits a Random Access Preamble to the eNB to request a radio connection. Since none of the CCCH, DCCH, or DTCH channels are enabled, the baseband decodes only PDSCH packets associated with the Random Access RNTI (RA-RNTI) to detect the RAR from the eNB. Packets identified by other RNTI types, such as the Cell RNTI (C-RNTI), are typically ignored.

State 2. This state is activated once the baseband receives RAR from the eNB. Following the RAR, the CCCH logical channel becomes active, and the baseband uses the C-RNTI assigned by the previous RAR message to decode upcoming packets. As only CCCH is activated in this state, the baseband ignores any packets mapped to DCCH-1, DCCH-2, or DTCH.

State 3. This state begins when the baseband receives the RRC Connection Setup message, which activates the DCCH-1 channel for transferring signaling messages during the AKA procedure. During this state, the baseband and eNB establish a secure connection with both NAS and Access Stratum (AS) security contexts. The baseband ignores any packets directed to DCCH-2 or DTCH, as these channels remain inactive.

State 4. After the RRC Connection Reconfiguration message is received, the baseband enters this state where DCCH-2 and

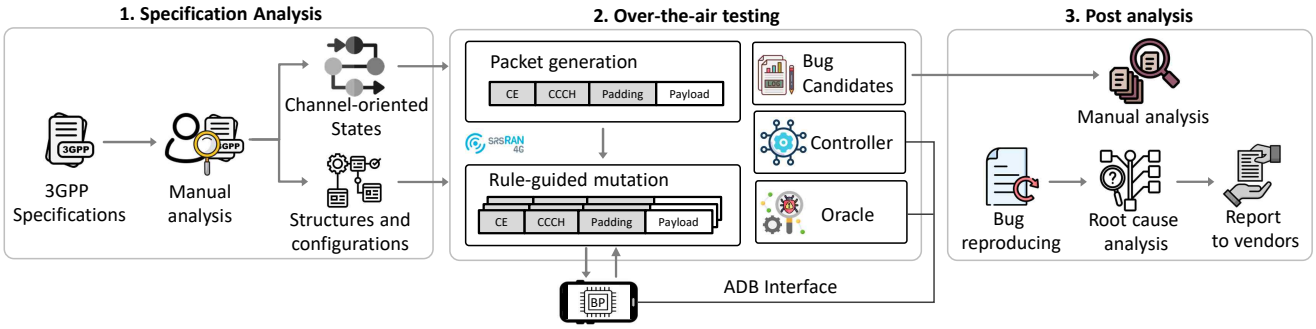


Figure 6: LLFUZZ design.

DTCH channels become active. Layer-specific configurations for the RLC and PDCP layers are also delivered in this RRC message. To this end, the baseband is fully configured to exchange both user data and signaling messages with the base station.

5.3 Identify Lower-Layer Packet Structures

While earlier studies [57] suggested that layer 2 messages are too short to be effectively fuzzed, our specification analysis reveals that lower-layer packet structures are significantly more complex and diverse than previously assumed. Table 2 provides a summary of supported packets by LLFUZZ. The following section describes important details for these layers. Due to space constraints, more details can be found in our repository (`https://github.com/SysSec-KAIST/LLFuzz`).

MAC layer. The MAC layer packet structures vary based on the baseband state during the Attach procedure (Figure 4). In the initial state (State 1), the baseband uses the RAR structure, transitioning to the DL-SCH structure in States 2-4 after decoding the RAR message. Also, the MAC layer contains 14 MAC CEs serving various functions, such as CA and power control. Nine of these have dedicated sub-payloads [4].

RLC layer. The structure of RLC packets is determined by the RLC mode (TM, UM, or AM) and the configurations of the Length Indicator (LI) and Sequence Number (SN) fields. These configurations are conveyed to the UE through RRC signaling messages during the Attach procedure. Since logical channels are not always active, LLFUZZ needs to determine the correct logical channel and corresponding PDU structures to test, as specified by the MAC layer sub-header. Totally, we identified 18 PDU structures in the RLC layer for testing.

PDCP layer. The PDCP packet structures differ between the control and user planes, represented by SRBs and DRBs, respectively. Similar to the RLC layer, configurations for these bearers (*e.g.*, SN field length) are delivered via RRC signaling during the Attach procedure. Additionally, the PDCP layer supports several dedicated control PDUs (*e.g.*, control PDUs for Ethernet Header Compression feedback) that report the status of header compression and Wireless Local Area Network (WLAN) aggregation protocols. These control PDUs are distinguished by the PDU type fields in the PDCP header. In total, LLFUZZ supports testing 17 PDU structures in the PDCP layer.

PHY layer. In the PHY layer, the UE expects different DCI and RNTI formats depending on the state of the baseband, duplexing techniques (*i.e.*, FDD/TDD), channel bandwidth, transmission mode, and carrier aggregation. In state 1, the UE uses DCI format 1A scrambled with the RA-RNTI. In subsequent states, DCI formats scrambled with the C-RNTI are expected. Specifically, in state 2, the UE typically uses DCI format 1A with the C-RNTI because the antenna configuration has not yet been received. In states 3 and 4, the DCI formats 1, 2A, or 2 (depending on the TM) are used, and scrambled with the C-RNTI. Our work supports 11 DCI structures, as shown in Table 2.

Note that the PHY layer consists not only of DCIs but also other signal processing blocks, such as Fast Fourier Transform (FFT), channel estimation, synchronization, and turbo decoding. These blocks have fixed input and output lengths and are typically processed in hardware components. Therefore, it is not relevant to discover memory corruption bugs in baseband, making them out of the scope in this work.

5.4 Test Case Generation

In the following, we describe how LLFUZZ generates test cases for lower layers based on specification analysis. Overall, the test case generation process consists of five main steps below. More details for each layer are provided in Appendix D.

Initial packet generation. For each layer, LLFUZZ selects a packet structure from the specification, generates valid packets, and uses them as seeds for subsequent mutation steps. To enhance test case diversity, LLFUZZ generates packets with varying numbers of components, particularly for structures that support multiple sub-headers, sub-payloads, or data chunks. For instance, in the MAC layer, LLFUZZ generates packets with 1, 3, or more than 50 sub-headers/sub-payloads. Several fields (*e.g.*, `F`, `F2`, `E`) are fixed at this step, as they pertain to the packet format and the number of components.

Packet truncation. LLFUZZ truncates the generated packets to create various test cases with valid and malformed header-payload combinations. For instance, it generates test cases containing only sub-headers without sub-payloads, mis-

Table 2: Supported structures in MAC, RLC, PDCP, and PHY layers.

Layer	No.	Packet structure	Sub-header format/Configuration	State	Logical channel
MAC	1–2	RAR PDUs [4, pp. 110–112]	RAR sub-header formats R1, R2	S1	-
	3–10	DL-SCH PDUs [4, pp. 88–89]	DL-SCH sub-header formats A, B, C, D, eA, eB, eC, eD	S2, S3, S4	All LCIDs
	11–19	14 MAC CEs⋆ [4, pp. 89–110]	DL-SCH sub-header format D	S2, S3, S4	Dedicated LCIDs for CEs
RLC	1–4	UM Data PDUs [5, pp. 24–26]	No LI / 11-bit LI, 10-bit SN / 5-bit SN	S4	DTCH
	5–6	AM Data PDUs [5, pp. 27–29]	No LI / 11-bit LI, 10-bit SN	S3, S4	DCCH-1, DCCH-2, DTCH
	7–10	AM Data PDUs [5, pp. 27–29]	No LI / 11-bit LI / 15-bit LI, 10-bit SN / 16-bit SN	S4	DTCH
	11–12	AM Data PDU Segments [5, pp. 30–33]	No LI / 11-bit LI, 10-bit SN	S3, S4	DCCH-1, DCCH-2, DTCH
	13–16	AM Data PDU Segments [5, pp. 30–33]	No LI / 11-bit LI / 15-bit LI, 10-bit SN / 16-bit SN	S4	DTCH
	17–18	Status PDUs [5, pp. 33–34]	10-bit SN / 16-bit SN	S3, S4	DCCH-1, DCCH-2, DTCH
PDCP	1	Data PDU for SRB [6, p. 38]	Fixed configuration	S3, S4	DCCH-1, DCCH-2
	2–5	Data PDUs for DRB [6, pp. 39, 42, 43]	7-bit SN / 12-bit SN / 15-bit SN / 18-bit SN	S4	DTCH
	6–8	Status reports [6, pp. 40–41]	12-bit SN / 15-bit SN / 18-bit SN	S4	DTCH
	9–11	Header compression feedbacks	ROHC*, EHC†, UDC‡ [6, pp. 40, 46, 47]	S4	DTCH
	12–14	WLAN aggregation status reports	12-bit SN / 15-bit SN / 18-bit SN [6, p. 44]	S4	DTCH
	15–17	WLAN aggregation end-markers	12-bit SN / 15-bit SN / 18-bit SN [6, p. 45]	S4	DTCH
PHY	1	DCI format 1A [2, p. 186]	RA-RNTI, FDD	S1	-
	2	DCI format 1A [2, p. 186]	C-RNTI, FDD	S2	-
	3–8	DCI formats 1/2/2A [2, pp. 184, 193, 197]	C-RNTI, FDD, TMs 1/2/3/4, Allocation Types 0/1	S3, S4	-
	9	DCI format 1A [2, p. 186]	PDCCH order	S4	-
	10	DCI format 1C [2, p. 190]	RA-RNTI, FDD	S1	-
	11	DCI format 0 [2, p. 177]	C-RNTI, FDD, Allocation Type 2	S3, S4	-

⋆ Nine of which have payloads [4].
∗ Robust Header Compression.
† Ethernet Header Compression.
‡ Uplink Data Compression.

matched numbers of sub-headers and sub-payloads (*e.g.*, 3 sub-headers but only 2 sub-payloads), or very short packets with just 1 or 2 bytes. In the PHY layer, DCI messages are treated as complete headers and are directly mutated in the next step without further modifications.

Header mutation. This step focuses on mutating the remaining fields within the sub-headers, excluding the fields already fixed in the previous steps (*e.g.*, `E, F2, F` fields). LLFuzz follows the specification to select appropriate mutation values for each field, ensuring changes do not invalidate adjacent fields. Particular attention is given to fields that indicate payload length, such as the `L` fields in the MAC layer and the `LI` fields in the RLC layer. For these fields, LLFuzz uses boundary values specified in the specification or large values that exceed the total length of the sub-payloads or data chunks. In the MAC layer, LLFuzz mutates the critical `LCID` field, which identifies the logical channel, to ensure a diverse mix of data types (*e.g.*, signaling, user data, and MAC CEs) within a single test case. As the PHY layer lacks explicit length indicator fields unlike other layers, LLFuzz focuses on generating test cases with reserved or abnormal values. For instance, in DCI format 2A, LLFuzz can set the Resource Block field to the maximum value of the bandwidth, which is abnormal for a single UE.

Payload mutation. Then, LLFuzz mutates payloads that are decoded and processed within the target testing layer. For instance, in the MAC layer, LLFuzz focuses on mutating RAR sub-payloads and MAC CEs, as these contain control information processed directly by the MAC layer. Similarly, in the RLC and PDCP layers, LLFuzz mutates Status PDUs and control PDUs associated with various header compression algorithms. For these payloads, LLFuzz relies on their structures as defined in the specifications to determine mutation values, ensuring meaningful and systematic testing. Notably, LLFuzz skips mutating payloads for upper layers, as they are beyond its scope.

Mapping test cases to the correct logical channel. A key challenge in testing the RLC and PDCP layers is ensuring test cases reach the correct logical channel. If packets are sent to inactive or incorrectly mapped channels, they will be ignored or improperly decoded by the baseband. To address this, LLFuzz tracks channel-oriented states to identify active channels and determine the necessary packet structures for data delivery. It then generates appropriate headers for the layers beneath the testing layer. For instance, when testing SRB1 in the PDCP layer (corresponding to the DCCH-1 logical channel), LLFuzz encapsulates test cases into an RLC AM Data PDU and further into a MAC PDU with `LCID 00001`, which transports RLC AM PDUs in DCCH-1. This hierarchical mapping ensures test cases are accurately delivered to the target logical channel and processed effectively.

5.5 Over-The-Air Testing

After generating test cases tailored to specific configurations and states, LLFuzz sends these test cases to the baseband over the air interface. For that, first, LLFuzz triggers the Attach procedure between the baseband and the framework's eNB to transition the baseband to the desired state. Next, it delivers the target configuration to the baseband using the RRC Connection Reconfiguration message. Once the baseband moves

to the target state with the target configuration, LLFuzz sends the corresponding test cases to the baseband. During this process, it monitors the ADB logcat output to detect crashes, as described in §5.6. If a crash is detected, LLFuzz stores the recent test cases as bug candidates for post analysis (§5.7). If not, LLFuzz disconnects the UE from the eNB to reset the baseband and begins the next iteration.

State management. To identify the baseband's current state, LLFuzz monitors all signaling messages exchanged between the baseband and the eNB during the Attach procedure. This information is essential to ensure the baseband is in the correct state before receiving any test cases. However, the baseband occasionally terminates the Attach procedure without progressing to the next state due to improper handling of our test cases, even though it does not crash. To address this, LLFuzz uses a dedicated timer for each state. If the timer expires and the baseband fails to progress to the next state, LLFuzz assumes it is disconnected and toggles airplane mode via ADB commands to recover the basebands.

Test various layer-specific configurations. LLFuzz uses RRC Connection Reconfiguration messages to deliver target configurations to the baseband during the Attach procedure. Specifically, in the RLC layer, LLFuzz configures the RLC mode (*i.e.*, AM, UM) and length of SN and LI fields within the RLC PDUs. For the PDCP layer, it configures the length of the SN field before sending the test cases. For DCI fuzzing, we first configure the eNB with a specific duplexing technique (FDD), channel bandwidth, and transmission mode. Then, LLFuzz uses the RRC Connection Setup to deliver these configurations to the UE. This step is essential for the UE to determine which DCI formats it should decode for communication.

Handle unsupported configurations in srsRAN. While the specifications define numerous configurations in the lower layers, srsRAN [53], which is the open-source LTE base station used in our framework, only supports a subset of them. For example, srsRAN supports only 10-bit sequence numbers (SN) in RLC AM Data DPUs but not 16-bit SN. To test this unsupported configuration, LLFuzz uses the following trick. LLFuzz first uses the RRC Connection Reconfiguration message, making the UE use the target configuration. Then, LLFuzz sends test cases immediately after the baseband successfully processes the RRC Connection Reconfiguration message. Subsequently, the baseband will process the test cases with the intended configuration before it notifies any errors due to the lack of eNB support for these configurations. We confirmed that the baseband processed the intended target configuration by observing the RRC Connection Reconfiguration Complete message sent by the UE, as captured in both the base station and XCAL log messages [59]. According to the RRC specification [7], the UE transmits this message upon successfully applying the configuration received from the eNB. In this way, we can work around the limitations of srsRAN and test unsupported configurations with a small engineering effort.

5.6 Oracle for Detecting Crashes

LLFuzz utilizes ADB radio logcat output to detect baseband crashes. Since the previous crash indicator [24] (*e.g.*, "CP Crash") is no longer applicable to recent basebands, we analyze the ADB logcat output during our testing to identify new crash log messages. To achieve this, we enable the manufacturer's debug mode on the baseband and send test cases until a crash occurs. We then focus on examining debug messages classified as Error or Fatal that appear immediately after receiving malicious packets. Also, we observed that the number of log messages significantly decreases during a crash until the baseband fully recovers. To avoid false positives, we repeat tests and only consider log messages that appear consistently across runs. Notably, the crash log messages vary across basebands from different vendors. For Qualcomm, MediaTek, and Samsung basebands, crashes are both indicated by "RADIO_OFF_OR_UNAVAILABLE" or "Modem Reset" debug messages. Interestingly, for Google Tensor basebands, the crash log message is uniquely identified as "Everybody panic!", distinguishing it from other vendors.

To validate crashes detected by the oracles, LLFuzz stores a set of recent test cases in a bug candidates. These cases are later verified using the manufacturer's debug mode (§5.7).

5.7 Post Analysis

To verify crashes, we manually reproduce bug candidates using manufacturer's debug mode. On Samsung devices, we enable this mode by dialing `*#9900#`, which shows a black screen immediately when the baseband crashes. This method is more reliable than ADB-based detection but requires manual rebooting after crashes, making it unsuitable for automated testing. During reproduction, we send test cases at a slower pace (one every 2 seconds) to prevent overlapping causes and eliminate false positives. If the debug mode is unavailable, we verify crashes by sending the stored test cases multiple times and monitoring the ADB logcat output, as described in §5.6.

Once crashes are verified, we further investigate their root causes by analyzing the structure of the test cases. To facilitate this analysis, we incorporate the state, channel, and configuration used during testing. We also test the verified cases across different baseband states and logical channels to thoroughly assess the impact of each crash. Finally, we report the root causes and impacts of the crashes to the baseband vendors.

6 Implementation

We implement LLFuzz on top of srsENB [53], an open-source, full-stack LTE base station. LLFuzz is written in C++ and consists of over 11,568 lines of code. Since the original C/C++ packet structures in srsENB are designed to store only legitimate data (*i.e.*, packets with correct structures), we

developed custom structures for test cases. Specifically, we abstracted most packet structures across the PHY, MAC, RLC, and PDCP layers, as described in §5.4, into custom C++ structures. In addition, we developed custom mutators to modify fields related to the headers and payloads of the targeted protocol layers. The mutated test cases are subsequently assembled into packets at the byte level using a custom packet generator and then transmitted to the target baseband. In the following, we describe noteworthy details in the implementation of LLFuzz.

Flexible DCI allocation. Our test cases for the MAC layer contain those with very small sizes (*i.e.*, 2 bytes). However, the default algorithm of srsENB only supports allocating radio resources in multiples of several Resource Blocks (RBs), which are typically larger than the size of these test cases. To address this, we develop a custom DCI allocation mechanism that enables the allocation of radio resources in a more flexible manner. In particular, LLFuzz utilizes both Resource Allocation Type 1 and Type 2, allowing finer-grained radio resource allocation. It then refers to the Transport Block Size table in the PHY layer specification [3] to select the appropriate Modulation and Coding Scheme, as well as the number of RBs based on the size of the test case. The remaining DCI fields are left unchanged as generated by srsENB, ensuring that the UE can correctly decode the message. This allows LLFuzz to support a broader range of test cases, including those with very small sizes.

Separated ADB thread. Reading radio logcat data from the ADB interface or sending commands to it is typically time-consuming. This is because it may take some time for devices to respond upon receiving commands. Such delays can block the main thread of the eNB, which requires real-time processing to function effectively as an LTE base station. To address this, LLFuzz implements a parallel thread to handle ADB communication in the srsENB, thereby preventing performance bottlenecks. This ADB thread is responsible for managing tasks such as sending airplane mode commands to the UE and retrieving radio logcat output to detect crashes. Additionally, it implements a function to reboot the UE in cases where the UE becomes permanently disconnected from the eNB due to unforeseen issues.

7 Evaluation

To evaluate LLFuzz, we aim to answer the following questions:

- **Q1:** How effective is LLFuzz in terms of finding memory corruption vulnerabilities on different layers and states? (§7.2)
- **Q2:** How efficiently does LLFuzz generate and test? (§7.3)
- **Q3:** Can LLFuzz's approach be adapted to 5G lower layers? (§7.4)
- **Q4:** How is LLFuzz better than existing tools? (§7.5)

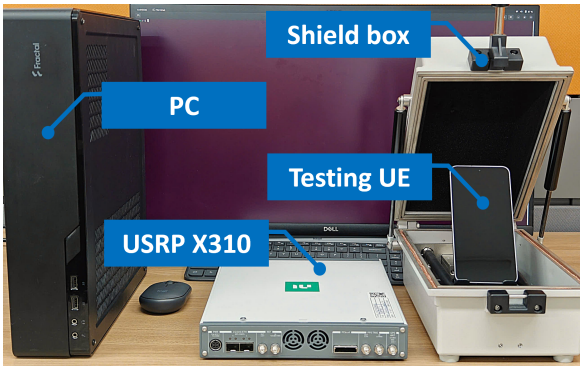


Figure 7: Evaluation setup.

7.1 Evaluation Setup

Figure 7 illustrates our over-the-air evaluation setup. We use a PC equipped with an Intel i7-11700K CPU and 32GB RAM to run the LLFuzz framework on Ubuntu 18.04. The PC is connected to a USRP X310 [55], which serves as the RF frontend for transmitting and receiving signals to and from the testing baseband. Additionally, the UE is connected to the PC via a USB cable to enable the ADB interface. To prevent malicious signals from interfering with other cellular devices, we place the UE inside a shield box (*i.e.*, Faraday cage) to isolate it from the external environment.

7.2 Bug Discovery

We evaluated 15 cellular basebands from five vendors: Qualcomm Snapdragon, Samsung Exynos, MediaTek, Google Tensor, and Huawei Kirin. A full list of tested devices and baseband models is available in Table 4 in the Appendix. In total, we identified nine memory corruption vulnerabilities: five in the MAC layer, two in the RLC layer, and two in the PDCP layer (Table 3). Below, we discuss LLFuzz's effectiveness in discovering memory corruption vulnerabilities in lower layers. In §8, we provide a detailed analysis of the discovered vulnerabilities.

Coverage. Using LLFuzz, we generated a total of 121,433 test cases for the lower layers of the LTE protocol stack, including 5,350 for PHY, 96,141 for MAC, 17,784 for RLC, and 2,158 for PDCP. These test cases cover all packet structures identified in §5.3.

Stateful testing. LLFuzz discovered vulnerabilities in various states of the lower layers, as summarized in Table 3. For instance, bug B2 in the Qualcomm baseband exclusively affects State 1, where the baseband decodes only the RAR messages. To trigger this bug, LLFuzz must generate test cases adhering to the RAR structure and deliver them to the baseband precisely while it is in State 1. If the same test cases are sent in other states, the baseband will not crash, as it expects a different structure in those states. Another example is bug B4, identified in the Helio P65 and G80 MediaTek basebands. Interestingly, this bug is limited to State 3 and does

Table 3: Discovered vulnerabilities across different vendors and protocol layers.

No.	Vendor*	Layer	Description	State	Configuration	Disclosure
B1	Qualcomm	MAC	Incorrect handling of the length field in the MAC header when a CCCH sub-header is present	S2, S3, S4	-	CVE-2025-21477 Patched
B2	Qualcomm	MAC	Incorrect handling of RAR messages containing only sub-headers without any payload	S1	-	CVE-2024-23385 Patched
B3	Qualcomm	RLC	Incorrect handling of the extension part of the RLC UM Data PDU header	S4	UM, 5-bit SN	Under review
B4	MediaTek	MAC	Incorrect handling of zero value in the length field of the MAC sub-header	S3	-	CVE-2024-20076 Patched
B5	MediaTek	MAC	Incorrect handling of MAC PDUs with many MAC CE sub-headers	S2, S3, S4	-	CVE-2024-20077 Patched
B6	MediaTek	MAC	Incorrect handling of continuously malformed MAC PDUs during the attach procedure	S2, S3	-	Affects only older firmware versions
B7	MediaTek	PDCP	Incorrect handling of 5-byte PDCP Data PDUs for the control plane	S4	-	CVE-2025-20659 Patched
B8	Tensor, Exynos	RLC	Incorrect handling of RLC AM Data PDUs containing many data chunks	S3, S4	AM, 11-bit LI, 10-bit SN	Under review
B9	Exynos	PDCP	Incorrect handling of 1-byte PDCP Data PDUs for the user plane	S4	12-bit SN	Under review

∗ A full list of tested devices and affected baseband models is available in Table 4 in the Appendix.

not affect the basebands in State 2 or State 4, despite these states utilizing similar MAC packet structures. This behavior suggests that the basebands may use distinct code paths to process MAC packets in different states, further emphasizing the importance of LLFuzz's stateful testing approach.

In addition, LLFuzz uncovered several memory corruptions, such as B1, B5, and B8, which are active across multiple states. These vulnerabilities are particularly critical because they can be exploited in multiple states, especially after the AKA procedure, when a secure context has already been established between the UE and the eNB. Notably, traditional memory corruptions at layer 3 tend to be less severe in this state due to the cryptographic protection of the data. These findings demonstrate LLFuzz's effectiveness in identifying memory corruption vulnerabilities across different sub-layers of the LTE protocol stack. Moreover, they highlight LLFuzz's scalability and robustness, as it successfully uncovered vulnerabilities across basebands from multiple vendors.

Support for various low-level layers. LLFuzz was able to identify memory corruption vulnerabilities across all sub-layers of layer 2 (*i.e.*, MAC, RLC, and PDCP). These results demonstrate LLFuzz's capability to systematically test the lower layers and uncover bugs induced by diverse layer-specific configurations.

Security implications. Lower layer bugs pose a significant security threat because they remain exploitable even after the AKA procedure. This is because these layers lack cryptographic protection for headers and layer-specific control messages (*e.g.*, RAR grants, MAC CEs, Status PDUs). Thus, all bugs discovered by LLFuzz in State 4 are vulnerable to over-the-air exploitation. For instance, an attacker can use the SigOver [60] or MitM [50] attacks to exploit bugs such as B7 or B9 in the post-AKA state without requiring the secret key. In addition, bugs in States 1 to 3 can be exploited using the FBS [36] attack model. Moreover, a skilled attacker can potentially escalate these exploits to achieve remote code execution by crafting specifically tailored malicious packets [10, 23].

7.3 Performance

To evaluate the performance of LLFuzz, we measured the number of test cases generated and tested within a 60-second interval. LLFuzz can send 1–5 test cases per session (*i.e.*, per Attach procedure), depending on the fuzzing state. Under stable conditions—when the UE safely drops malformed test cases and completes the attach procedure—LLFuzz achieves a throughput of up to 220 test cases per 60 seconds. In such cases, LLFuzz leverages signaling messages (*e.g.*, RRC Connection Release and Paging) to initiate new testing sessions and maximize testing speed. However, this speed is significantly reduced when the baseband crashes or silently disconnects without crashing. In these situations, LLFuzz must toggle Airplane mode via ADB to recover the baseband and re-establish the radio connection. In our evaluation, the recovery time after a crash varied between 10 and 15 seconds, depending on the baseband model and crash type. We also observed that basebands may permanently disconnect from the framework after failing to complete the attach procedure more than five times. In such cases, manual intervention is required to restart both the baseband and the base station to restore the connection.

Testing a single UE with LLFuzz typically took between 42.5 and 58.5 hours. Exynos basebands (*e.g.*, Galaxy S24) required approximately 58.5 hours, while MediaTek basebands (*e.g.*, Xiaomi K40 Gaming) took around 42.5 hours.

7.4 Adapting LLFuzz for 5G Basebands

While LLFuzz was initially developed for LTE, its approach can be extended to 5G basebands, which are based on the

same design principles and share similar protocol stacks and lower-layer channel structures. As a proof of concept, we implemented a minimal version of LLFuzz for 5G lower layers, referred to as LLFuzz-5G.

LLFuzz-5G is built on top of the srsRAN 5G project [54], a separate codebase from srsRAN_4G [53], requiring substantial adaptation effort. We ported key components of LLFuzz —including stateful testing, logical channel-driven design, configuration-aware testing—and developed a grammar-based test case generator to support the 5G PDCP layer.

Target Layers and Structures. We target the 5G PDCP layer, which includes multiple logical channels (*e.g.*, DCCH-1, DCCH-2, and DTCH) and exhibits stateful behavior like LTE (Figure 4). We generated test cases for two PDCP structures: (1) Data PDUs for SRBs with fixed configurations, and (2) Data PDUs for DRBs configured with a 12-bit SN. Among our test devices, we used a Xiaomi K40 Gaming that supports connection to an open-source gNB to transmit the test cases.

Results. Using LLFuzz-5G, we discovered two previously unknown bugs. The first affects PDCP Data PDUs for SRBs and, unlike LTE bug B7, only occurs in State 3, where the DCCH-1 logical channel is used to deliver signaling messages before the AKA procedure. Interestingly, the second bug impacts the RRC layer, as our PDCP test case also includes an embedded RRC payload. Notably, prior work such as 5Ghoul [20] failed to uncover these issues. As these bugs were identified and reported to the vendor recently, we cannot disclose their details in this paper. Full details will be provided in our open-source release upon vendor confirmation. These findings demonstrate that the LLFuzz approach can be effectively extended to fuzz and uncover vulnerabilities in the 5G lower layers.

7.5 Comparison with Existing Tools

We compare LLFuzz with 5Ghoul, the only publicly available tool that supports testing the lower layers of basebands. To this end, we analyzed their source code and also ran 5Ghoul in our environment to analyze their test case generation methods. By default, 5Ghoul uses default packets from the open-source OAI gNB as mutation seeds. It selects a packet, parses it using the Wireshark's packet dissector, and randomly mutates fields across layers (*e.g.*, MAC, RLC, PDCP, RRC, NAS). By comparing these test cases with those generated by LLFuzz, we identified several limitations in 5Ghoul's approach.

Limited layer-specific configurations. 5Ghoul tests only the default configuration from the open-source gNB, limiting packet structures to those in the default OAI setup. In contrast, LLFuzz uses a configuration-aware approach to explore a broader range of configurations and achieve better structural coverage. For instance, using only the default srsENB configuration would fail to trigger bug B3 in LTE, requiring a 5-bit SN in RLC UM. Moreover, LLFuzz can actively configure the baseband using RRC Reconfiguration messages to test unsupported configurations (§5.5).

Limited test case diversity. As legitimate packets usually contain only a few sub-headers, sub-payloads, and standard lengths (*e.g.*, more than 9-byte PDCP packets), 5Ghoul's mutation approach struggles to generate diverse test cases, particularly malformed packets with many sub-headers (> 50) or very short lengths. This limitation explains why 5Ghoul failed to detect our two new bugs in 5G PDCP and RRC, which require very short test cases rarely found in open-source base stations. On the other hand, LLFuzz leverages a specification-guided approach to generate such test cases effectively.

Untargeted logical channel. Random field mutation in 5Ghoul may unintentionally alter essential fields, such as LCID or RLC SN, leading to test cases that are not correctly routed to the target logical channel. For example, when analyzing the first 500 test cases generated by 5Ghoul using Wireshark, we observed that more than 50% had incorrect LCIDs. These test cases were likely rejected by the baseband before reaching the RLC layer. The rejection rate could be even higher due to incorrect RLC headers. In contrast, LLFuzz adopts a logical channel-driven and specification-guided approach to generate test cases tailored to specific logical channels, ensuring they are correctly routed to the target channel.

8 Case Study

In this section, we discuss two out of nine bugs discovered by LLFuzz. Due to space constraints, additional case studies are provided in Appendix C.

B2: Incorrect handling of RAR messages containing all sub-headers without any payload. We identified a vulnerability that affects several Qualcomm basebands in State 1. According to the MAC specification [4], RAR messages can contain multiple sub-headers, each with a Random Access Preamble Identifier (`RAPID`) field to identify the intended UE. To trigger the bug, we crafted a malformed RAR message with 57 sub-headers, placing the `RAPID` matching the target UE in the final sub-header while using different `RAPIDs` for all others. Further analysis revealed that the crash occurs only when the matching `RAPID` sub-header is at the end of the message. We reported this issue to Qualcomm, which assigned CVE-2024-23385 with a high severity rating.

B3: Incorrect handling of the extension part of the RLC UM Data PDU header with 5-bit Sequence Number. We identified a vulnerability in several Qualcomm basebands related to the extension part of the RLC UM Data PDU. According to the specification, these packets can contain multiple data chunks indicated by [`E`, `LI`] field pairs in the header extension. For example, a PDU with three data chunks would have two [`E`, `LI`] pairs, as the last chunk's size is determined by the remaining bytes. To exploit this bug, we created a malformed 2-byte packet with a 1-byte header and 1-byte extension part, setting the E field to 1 (indicating another [`E`, `LI`] pair should follow) despite having no space for it. The likely root cause

of this bug is insufficient validation of the extension part's length before parsing. This issue occurs only when the RLC UM Data PDU is configured with a 5-bit Sequence Number and an 11-bit Length Indicator in State 4.

9 Discussion and Limitations

Supporting 5G protocol. LLFuzz primarily targets LTE due to its long-term presence in smartphone basebands—similar to 2G, which remains widely supported in modern devices (many studies continue to exploit 2G vulnerabilities). However, we believe that the techniques and approaches employed in LLFuzz can be adapted to 5G protocols, as the 5G lower layers share a similar design with LTE. As a proof of concept, we developed a minimal version of LLFuzz that supports two structures in the 5G PDCP layer, through which we uncovered two previously unknown bugs (§7.4).

Despite this success, we encountered several practical challenges during our 5G development. First, most of our 5G devices failed to connect to the open-source gNBs. For these devices, the UEs did not even initiate the Random Access Procedure, making further debugging impossible. We experimented with different bands, bandwidths, and MNC/MCC settings, but all attempts failed. A few devices (*e.g.*, Xiaomi K40 Gaming, Google Pixel 8 Pro) were able to connect using manual NR forcing (*e.g.*, enabling NR-only mode via `*#*#4636*#*#`). Second, automatic testing posed additional challenges. For instance, when testing the Xiaomi K40 Gaming, we observed that the device automatically reset its connection mode to 3G/LTE/NR after crashing. As a result, we had to manually reconfigure the device to NR-only mode after each crash, significantly slowing down the testing process. Third, the 5G lower layers are inherently more complex than LTE. For example, the 5G MAC layer includes over 20 MAC CEs with complicated structures. This added complexity requires substantial effort to fully support 5G protocols.

Fuzzing DCI messages. Although this work did not identify memory corruptions in DCI, such vulnerabilities could arise if DCI messages are improperly handled. The PHY layer must translate DCI contents into uplink/downlink grants and control information (*e.g.*, TPC), using indices to retrieve values from predefined tables. Programming errors in this process—especially when handling reserved or abnormal values—could result in memory corruption. For instance, LTE DCI format 2 includes a 3-bit field for precoding information, which determines the precoding matrix. However, the value 7 in this field is reserved for future use; incorrect access to this value might lead to memory corruption. Given the potential for such vulnerabilities, LLFuzz supports fuzzing six DCI formats (Table 2). While the specification defines additional DCI formats, we focused only on those supported by the srsENB. Several unsupported formats are specific to NB-IoT, which is outside the scope of this work. Moreover, testing many DCI formats requires a more complex setup, such as synchronized base stations and MIMO antennas. We leave the exploration of these DCI structures for future work.

Vendor-specific behaviors. Basebands from the same vendor typically exhibit similar behaviors during our tests. As a result, vulnerabilities discovered on one device often affect other devices from the same vendor (Table 4). However, we observed notable differences between Exynos and Google basebands, even though the latter is reportedly based on the Exynos. For example, while both bugs B8 and B9 were found in the Exynos modem, only B8 affected the Google Pixel 6a.

Limitations. First, like other over-the-air fuzzing tools, LLFuzz cannot identify the root cause of crashes at the memory level. For instance, it cannot determine whether a crash is caused by a buffer overflow or other types of memory access errors. Emulation-based approaches, such as FirmWire [24] or BaseSAFE [39], can be more helpful in terms of this. However, LLFuzz excels in rapidly detecting crashes and is universally applicable to any baseband device, regardless of vendor or implementation. Second, LLFuzz detects only memory corruptions that lead to a baseband crash or reset. Silent corruptions, which do not trigger a crash but may still be exploitable [40], are beyond its detection capability. In such cases, LLFuzz may be more useful to vendors with access to the baseband source code for further analysis. Third, as LLFuzz employs an ADB-based oracle to detect crashes and recover basebands, its support for iPhones is limited. However, since recent iPhone models are also equipped with Qualcomm basebands, we believe that some vulnerabilities discovered in Android devices can often be applicable to iPhones. For instance, we confirmed that the bug B1 discovered on a Samsung Galaxy Note 20 Ultra was also reproducible on an iPhone 13 Pro. We reported this result to Apple; they confirmed the issue and assigned CVE-2024-27874.

10 Related Work

Reverse engineering on basebands. Because the baseband operates as a black box, reverse engineering has been extensively applied to analyze baseband binaries [12, 13, 21, 22, 23, 29, 30, 33, 57, 58]. Basespec [30] used static analysis and symbolic execution to examine layer 3 decoding functions and compared the results against specifications to identify non-compliance. While reverse engineering is effective at detecting memory corruption vulnerabilities, it is labor-intensive and not scalable for multiple basebands.

Emulation-based approach for memory corruptions. To address the limitations of reverse engineering, another line of research proposed using emulation to dynamically analyze baseband firmware [24, 32, 39, 48]. FirmWire [24] successfully emulated Shannon and MediaTek basebands and leveraged the open-source fuzzer AFL++[18] to test layer 3 protocol functions. Building on FirmWire, Goos *et al.* [15] extended its capabilities to emulate layer 2 protocols on GSM basebands. Very recently, LORIS [48] and BaseBridge [32] enabled emulation-based stateful fuzzing for Samsung and

MediaTek basebands. Although effective at identifying root causes of memory corruption, prior work focused on layer 3 and GSM protocols. In contrast, LLFuzz targets layers 1 and 2 of the LTE stack, which are significantly harder to emulate due to their protocol complexity.

Over-the-air fuzzing for memory corruptions. Over-the-air fuzzing has emerged as an effective approach for discovering memory corruptions in cellular basebands [19, 20, 41, 46, 47, 56]. Most of these works, however, have focused on layer 3 protocols within the GSM protocol stack. Only a few efforts have targeted layer 2. Pestrea *et al.* [46] fuzzed the MAC layer of LTE base stations by generating a few test cases through combinations of MAC sub-header fields. More recently, 5Ghoul [19, 20, 52] targeted both layers 3 and 2 of the 5G protocol, but relied on random field mutations of legitimate messages generated by an open-source gNB. In contrast, LLFuzz adopts a more systematic and efficient approach, utilizing a specification-guided method to generate test cases. Its novel channel-driven, configuration-aware testing framework is tailored specifically for the lower layers, allowing systematic testing of diverse packet structures in multiple channels. Apart from cellular basebands, several previous works attempted to fuzz layer 2 of Bluetooth devices [8, 44]. While both target lower layers, LLFuzz addresses the unique challenges of the LTE protocol, including multiple logical channels and dynamically configurable packet structures.

Finding standard non-compliance bugs. Several studies have focused on identifying specification non-compliance bugs in cellular networks [11, 14, 28, 31, 43, 49]. Rupprecht *et al.* [49] proposed a framework to test encryption and authentication algorithms on the UE side. DoLTEst [43] introduced a negative testing approach to uncover non-compliance bugs in LTE basebands. Contester [14] leveraged natural language processing and machine learning to automatically generate conformant test cases for NAS protocol.

Testing other network components. Researchers have explored other cellular network components: RANsacked [9] targets RAN-Core interfaces, BaseMirror [35] focuses on Android's Radio Interface Layer, and Patir *et al.* [45] used LLMs to find bugs in open-source UEs—all distinct from our focus on closed-source commercial baseband components.

Attacks on lower layers. The absence of security mechanisms in lower layers has enabled various attacks, including overshadowing [16, 37, 60], user localization [34, 42], passive eavesdropping [17, 25, 38], and fake base stations [36]. LLFuzz reinforces the need for lower-layer protection by revealing multiple memory corruption vulnerabilities.

11 Conclusion

In this work, we introduced LLFuzz, an over-the-air testing framework designed to uncover memory corruption in the lower layers of cellular basebands. LLFuzz employs a logical channel-driven, configuration-aware testing approach to systematically explore the lower layers of basebands. Through specification-guided test case generation, LLFuzz created tailored test cases for a wide range of packet structures across the PHY, MAC, RLC, and PDCP layers. We evaluated LLFuzz on 15 commercial basebands from five major vendors, uncovering nine previously unknown memory corruptions. These findings highlight the critical importance of testing and securing the lower layers of cellular basebands, which are frequently underexamined yet essential for ensuring the reliability and security of mobile communication systems.

Acknowledgments

We sincerely appreciate reviewers and shepherd for their valuable comments. This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.RS-2024-00437252, Development of anti-sniffing technology in mobile communication and AirGap environments).

Ethical Considerations

Responsible Disclosure. We responsibly disclosed all our findings to the baseband vendors, providing detailed reports and engaging in discussions to assist their investigations. As a result, the status of disclosed vulnerabilities is as follows:

- Bugs B1 and B2 have already been patched by Qualcomm, and bugs B4, B5, and B7 have already been patched by MediaTek.
- Bug B6 affects only older baseband firmware versions and had already been patched in the latest firmware.
- Bugs B8, and B9 have been successfully verified, and vendors are working on releasing patches.
- Bug B3 is still under review, and we are actively assisting the vendor in reproducing it.

We continue to work closely with the vendors to ensure these issues are fully addressed.

Controlled Experiment. We conducted our experiment in a isolated environment (*i.e.* Faraday cage) to prevent malicious signals from affecting other cellular devices. Also, we operate our base station in a frequency band that is not used by commercial cellular networks. This setup ensures that our experiment does not violate any local regulations in our country.

Open Science

As outlined in §1, upon acceptance of this paper, we will make the source code of LLFuzz publicly available at https://github.com/SysSec-KAIST/LLFuzz. In addition, the source code will be archived in the artifact repository at https://doi.org/10.5281/zenodo.15602679. We hope that our framework will foster further research in this domain.

References

[1] libFuzzer – A Library for Coverage-guided Fuzz Testing. https://llvm.org/docs/LibFuzzer.html.

[2] 3GPP. TS 36.212, v16.2.0. LTE; Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and channel coding., 2020.

[3] 3GPP. TS 36.213, v16.2.0. LTE; Evolved Universal Terrestrial Radio Access (E-UTRA); Physical layer procedures., 2020.

[4] 3GPP. TS 36.321, v16.1.0. Evolved Universal Terrestrial Radio Access (E-UTRA); Medium Access Control (MAC) protocol specification., 2020.

[5] 3GPP. TS 36.322, v16.0.0. Evolved Universal Terrestrial Radio Access (E-UTRA); Radio Link Control (RLC) protocol specification., 2020.

[6] 3GPP. TS 36.323, v16.1.0. Evolved Universal Terrestrial Radio Access (E-UTRA); Packet Data Convergence Protocol (PDCP) specification., 2020.

[7] 3GPP. TS 36.331, v16.1.1. LTE; Evolved Universal Terrestrial Radio Access (E-UTRA); Radio Resource Control (RRC); Protocol specification., 2020.

[8] Pyeongju Ahn, Yeonseok Jang, Seunghoon Woo, and Heejo Lee. BloomFuzz: Unveiling Bluetooth L2CAP Vulnerabilities via State Cluster Fuzzing with Target-Oriented State Machines. In *European Symposium on Research in Computer Security*, 2024.

[9] Nathaniel Bennett, Weidong Zhu, Benjamin Simon, Ryon Kennedy, William Enck, Patrick Traynor, and Kevin RB Butler. RANsacked: A Domain-Informed Approach for Fuzzing LTE and 5G RAN-Core Interfaces. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[10] David BERARD and Vincent DEHORS. Zero-Click RCE on the Tesla Infotainment Through Cellular Network. *OffensiveCon*, 2024.

[11] Evangelos Bitsikas, Syed Khandker, Ahmad Salous, Aanjhan Ranganathan, Roger Piqueras Jover, and Christina Pöpper. UE Security Reloaded: Developing a 5G Standalone User-Side Security Testing Framework. In *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2023.

[12] Amat Cama. ASN.1 and Done: A Journey of Exploiting ASN.1 Parsers in the Baseband. *OffensiveCon*, 2023.

[13] Amat Cama. A walk with Shannon. *OPCDE*, 2018.

[14] Yi Chen, Di Tang, Yepeng Yao, Mingming Zha, XiaoFeng Wang, Xiaozhong Liu, Haixu Tang, and Baoxu Liu. Sherlock on Specs: Building LTE Conformance Tests through Automated Reasoning. In *Proceedings of the 32nd USENIX Security Symposium*, 2023.

[15] Goos Dyon and Muench Marius. Overcoming State: Finding Baseband Vulnerabilities by Fuzzing Layer-2. *BlackHat*, 2024.

[16] Simon Erni, Martin Kotuliak, Patrick Leu, Marc Roeschlin, and Srdjan Capkun. AdaptOver: Adaptive Overshadowing Attacks in Cellular Networks. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, 2022.

[17] Robert Falkenberg and Christian Wietfeld. FALCON: An Accurate Real-Time Monitor for Client-Based Mobile Network Data Analytics. In *IEEE Global Communications Conference*, 2019.

[18] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining Incremental Steps of Fuzzing Research. In *the 14th USENIX Workshop on Offensive Technologies*, 2020.

[19] Matheus E Garbelini, Zewen Shang, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. Towards Automated Fuzzing of 4G/5G Protocol Implementations Over the Air. In *IEEE Global Communications Conference*, 2022.

[20] Matheus E Garbelini, Zewen Shang, Shijie Luo, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. 5GHOUL: Unleashing Chaos on 5G Edge Devices. https://asset-group.github.io/disclosures/5ghoul/.

[21] Nico Golde and Daniel Komaromy. Breaking Band: Reverse Engineering and Exploiting the Shannon Baseband. *REcon*, 2016.

[22] Marco Grassi and Xingyu Chen. Exploring the MediaTek Baseband. *OffensiveCon*, 2020.

[23] Marco Grassi, Muqing Liu, and Tianyi Xie. Exploitation of a Modern Smartphone Baseband. *BlackHat*, 2018.

[24] Grant Hernandez, Marius Muench, Dominik Maier, Alyssa Milburn, Shinjo Park, Tobias Scharnowski, Tyler Tucker, Patrick Traynor, and Kevin Butler. FIRMWIRE: Transparent Dynamic Analysis for Cellular Baseband Firmware. In *Proceedings of the Annual Network and Distributed Systems Security Symposium*, 2022.

[25] Dinh Tuan Hoang, CheolJun Park, Mincheol Son, Taekkyung Oh, Sangwook Bae, Junho Ahn, BeomSeok Oh, and Yongdae Kim. LTESniffer: An Open-source

LTE Downlink/Uplink Eavesdropper. In *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2023.

[26] Byeongdo Hong, Shinjo Park, Hongil Kim, Dongkwan Kim, Hyunwook Hong, Hyunwoo Choi, Jean-Pierre Seifert, Sung-Ju Lee, and Yongdae Kim. Peeking Over the Cellular Walled Gardens - A Method for Closed Network Diagnosis. In *IEEE Transactions on Mobile Computing*, 2018.

[27] Syed Rafiul Hussain, Mitziu Echeverria, Omar Chowdhury, Ninghui Li, and Elisa Bertino. Privacy Attacks to the 4G and 5G Cellular Paging Protocols using Side Channel Information. In *Proceedings of the Annual Network and Distributed Systems Security Symposium*, 2019.

[28] Syed Rafiul Hussain, Imtiaz Karim, Abdullah Al Ishtiaq, Omar Chowdhury, and Elisa Bertino. Noncompliance as Deviant Behavior: An Automated Black-box Noncompliance Checker for 4G LTE Cellular Devices. In *Proceedings of the 28th ACM Conference on Computer and Communications Security*, 2021.

[29] Eunsoo Kim, Min Woo Baek, CheolJun Park, Dongkwan Kim, Yongdae Kim, and Insu Yun. BASECOMP: A Comparative Analysis for Integrity Protection in Cellular Baseband Software. In *Proceedings of the 32nd USENIX Security Symposium*, 2023.

[30] Eunsoo Kim, Dongkwan Kim, CheolJun Park, Insu Yun, and Yongdae Kim. BASESPEC: Comparative Analysis of Baseband Software and Cellular Specifications for L3 Protocols. In *Proceedings of the Annual Network and Distributed Systems Security Symposium*, 2021.

[31] 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*, 2019.

[32] Daniel Klischies, Dyon Goos, David Hirsch, Alyssa Milburn, Marius Muench, and Veelasha Moonsamy. BaseBridge: Bridging the Gap between Emulation and Over-The-Air Testing for Cellular Baseband Firmware. In *IEEE Symposium on Security and Privacy*, 2025.

[33] Daniel Komaromy. Basebanheimer: Now I Am Become Death, The Destroyer Of Chains. *OffensiveCon*, 2023.

[34] Martin Kotuliak, Simon Erni, Patrick Leu, Marc Röschlin, and Srdjan Čapkun. LTrack: Stealthy Tracking of Mobile Phones in LTE. In *Proceedings of the 31st USENIX Security Symposium*, 2022.

[35] Wenqiang Li, Haohuang Wen, and Zhiqiang Lin. BaseMirror: Automatic Reverse Engineering of Baseband Commands from Android's Radio Interface Layer. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[36] Zhenhua Li, Weiwei Wang, Christo Wilson, Jian Chen, Chen Qian, Taeho Jung, Lan Zhang, Kebin Liu, Xiangyang Li, and Yunhao Liu. FBS-Radar: Uncovering Fake Base Stations at Scale in the Wild. In *Proceedings of the Annual Network and Distributed Systems Security Symposium*, 2017.

[37] Norbert Ludant and Guevara Noubir. SigUnder: A Stealthy 5G Low Power Attack and Defenses. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2021.

[38] Norbert Ludant, Pieter Robyns, and Guevara Noubir. From 5G Sniffing to Harvesting Leakages of Privacy-preserving Messengers. In *IEEE Symposium on Security and Privacy*, 2023.

[39] Dominik Maier, Lukas Seidel, and Shinjo Park. BaseSAFE: Baseband Sanitized Fuzzing through Emulation. In *Proceedings of the 13th Conference on Security and Privacy in Wireless and Mobile Networks*, 2020.

[40] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *Proceedings of the Annual Network and Distributed Systems Security Symposium*, 2018.

[41] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[42] Taekkyung Oh, Sangwook Bae, Junho Ahn, Yonghwa Lee, Tuan Dinh Hoang, Min Suk Kang, Nils Ole Tippenhauer, and Yongdae Kim. Enabling Physical Localization of Uncooperative Cellular Devices. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, 2024.

[43] CheolJun Park, Sangwook Bae, BeomSeok Oh, Jiho Lee, Eunkyu Lee, Insu Yun, and Yongdae Kim. DoLTEst: In-depth Downlink Negative Testing Framework for LTE Devices. In *Proceedings of the 31st USENIX Security Symposium*, 2022.

[44] Haram Park, Carlos Kayembe Nkuba, Seunghoon Woo, and Heejo Lee. L2Fuzz: Discovering Bluetooth L2CAP vulnerabilities using stateful fuzz testing. In *Proceedings of the 52nd International Conference on Dependable Systems and Networks*, 2022.

[45] Rupam Patir, Qiqing Huang, Keyan Guo, Wanda Guo, Guofei Gu, Haipeng Cai, and Hongxin Hu. Towards LLM-Assisted Vulnerability Detection and Repair for Open-Source 5G UE Implementations. In *Proceedings of the Annual Network and Distributed Systems Security Symposium*, 2025.

[46] Anna Pestrea. Fuzz Testing on eNodeB Over the Air Interface: Using Fuzz Testing as a Means of Testing Security, 2021.

[47] Srinath Potnuru and Prajwol Kumar Nakarmi. Berserker: ASN.1-based Fuzzing of Radio Resource Control Protocol for 4G and 5G. In *17th International Conference on Wireless and Mobile Computing, Networking and Communications*, 2021.

[48] Ali Ranjbar, Tianchang Yang, Kai Tu, Saaman Khalilollahi, and Syed Rafiul Hussain. Stateful Analysis and Fuzzing of Commercial Baseband Firmware. In *IEEE Symposium on Security and Privacy*, 2025.

[49] David Rupprecht, Kai Jansen, and Christina Pöpper. Putting LTE Security Functions to the Test: A Framework to Evaluate Implementation Correctness. In *the 10th USENIX Workshop on Offensive Technologies*, 2016.

[50] David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. Breaking LTE on Layer Two. In *IEEE Symposium on Security and Privacy*, 2019.

[51] Jaeku Ryu. LTE Downlink Channel Mapping. *ShareTechnote*.

[52] Zewen Shang, Matheus E Garbelini, and Sudipta Chattopadhyay. U-Fuzz: Stateful Fuzzing of IoT Protocols on COTS Devices. In *IEEE Conference on Software Testing, Verification and Validation*, 2024.

[53] Software Radio Systems. srsRAN_4G. https://github.com/srsran/srsRAN_4G.

[54] Software Radio Systems. srsRAN_Project. https://github.com/srsran/srsRAN_Project.

[55] USRP X310. https://www.ettus.com/x310-kit/.

[56] Hongxin Wang, Baojiang Cui, Wenchuan Yang, Jia Cui, Li Su, and Lingling Sun. An Automated Vulnerability Detection Method for the 5G RRC Protocol Based on Fuzzing. In *4th International Conference on Advances in Computer Technology, Information Science and Communications*, 2022.

[57] Ralf-Philipp Weinmann. All Your Baseband Are Belong to Us. *Hack.lu*, 2010.

[58] Ralf-Philipp Weinmann. Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks. In *the 6th USENIX Workshop on Offensive Technologies*, 2012.

[59] Accuver XCAL. http://www.accuver.com/.

[60] Hojoon Yang, Sangwook Bae, Mincheol Son, Hongil Kim, Song Min Kim, and Yongdae Kim. Hiding in Plain Signal: Physical Signal Overshadowing Attack on LTE. In *Proceedings of the 28th USENIX Security Symposium*, 2019.

[61] Google Project Zero. Multiple Internet to Baseband Remote Code Execution Vulnerabilities in Exynos Modems. https://googleprojectzero.blogspot.com/2023/03/multiple-internet-to-baseband-remote-rce.html.

A Acronyms

AKA	Authentication and Key Agreement
AM	Acknowledged Mode
BCCH	Broadcast Control Channel
CCCH	Common Control Channel
CE	Control Element
DCI	Downlink Control Indicator
DCCH	Dedicated Control Channel
DRB	Data Radio Bearer
DTCH	Dedicated Traffic Channel
eNB	Evolved Node B (Base Station)
FDD	Frequency Division Duplex
GPRS	General Packet Radio Service
GSM	Global System for Mobile Communications
HARQ	Hybrid Automatic Repeat Request
IMEI	International Mobile Equipment Identity
LCID	Logical Channel ID
LI	Length Indicator
MAC	Medium Access Control
NAS	Non-Access Stratum
PCCH	Paging Control Channel
PDU	Protocol Data Unit
PDCP	Packet Data Convergence Protocol
PHY	Physical Layer
RAR	Random Access Response
RLC	Radio Link Control
RNTI	Radio Network Temporary Identifier
ROHC	Robust Header Compression
RRC	Radio Resource Control
SDU	Service Data Unit
SN	Sequence Number
SRB	Signaling Radio Bearer
TM	Transparent Mode
UE	User Equipment
UM	Unacknowledged Mode

B Tested Devices

The list of tested devices and their baseband versions is provided in Table 4.

C More Case Studies

C.1 B1: Incorrect Handling of the Length Field in the MAC Sub-Header

This vulnerability occurs when the baseband receives a malformed MAC DL-SCH PDU with a CCCH sub-header whose Length field exceeds the actual payload size. To trigger this bug, we crafted a MAC PDU with three sub-headers: `LCID 11100` (Contention Resolution CE), `LCID 00000` (CCCH) with the `L` field set to 127, and `LCID 11111` (padding). The payload contained only 6 bytes for the Contention Resolution CE and 2 bytes for CCCH—far less than the 127 bytes indicated by the CCCH sub-header. This vulnerability occurs only if the CCCH sub-header is present, and it affects multiple Qualcomm basebands in States 2–4.

C.2 B5: Incorrect Handling of MAC PDUs with Many MAC CE Sub-headers

The modem crashed after receiving a malformed MAC DL-SCH PDU with 50 sub-headers for MAC CEs and an invalid payload length. For the sub-headers, we set the `LCID` fields to `11100` (Contention Resolution), which has a fixed length of 6 bytes. However, the payload consisted of only 7 bytes of random data, far below the required 300 bytes for 50 MAC CEs. The reason for this crash appears to be the baseband failed to validate the actual payload length (7 bytes), resulting in a crash. Further analysis revealed that the baseband also crashes when other MAC CE LCIDs, such as `11011` (Activation/Deactivation), `11101` (Timing Advance), and `11110` (DRX Command), are used in the same manner. This vulnerability affects multiple MediaTek basebands in states 2–4. We reported the issue to MediaTek, which assigned it CVE-2024-20077 with a high severity rating.

C.3 B6: Incorrect Handling of Continuously Malformed MAC PDUs During the Attach Procedure

During our fuzzing tests, we observed an interesting behavior in several MediaTek basebands. When the baseband receives a malformed MAC DL-SCH PDU (*i.e.*, a PDU where the sub-payload length is less than the L field in the sub-header) during States 2 or 3 of the Attach procedure, it immediately terminates the current procedure and initiates a new one. In the new Attach procedure, the baseband exhibits the same behavior: it repeatedly terminates and initiates a new Attach procedure if the same malformed test case is sent. We found that continuously sending malformed MAC PDUs to force the baseband to trigger new Attach procedures 4–6 times eventually causes a crash. We reported this vulnerability to MediaTek, but it only affects older baseband firmware versions and had already been fixed in recent releases.

C.4 B7: Incorrect Handling of 5-byte PDCP Data PDUs for the Control Plane

We discovered a vulnerability in which basebands crash upon receiving PDCP Data PDUs that are shorter than expected. According to the PDCP specification [6], control plane Data PDUs must be at least 6 bytes long: a 1-byte header, a non-empty data field, and 4 bytes for integrity protection. By sending a malformed 5-byte PDU, we triggered crashes due to insufficient length validation, resulting in a buffer overread. Notably, PDUs shorter than 5 bytes (*e.g.*, 3 or 4 bytes) do not cause crashes. This vulnerability is specific to State 4 and does not affect State 3, despite both states using the same PDU structure for SRB1 and SRB2.

C.5 B8: Incorrect Handling of RLC AM Data PDUs Containing Many Data Chunks

We found several Samsung Exynos and Google Tensor basebands crash when receiving RLC AM PDUs with many data chunks. According to specifications, this RLC PDU can contain multiple data chunks described by `[E, LI]` field pairs. To trigger the vulnerability, we crafted a malformed PDU with 100 data chunks (each 10 bytes), encapsulated it into a MAC PDU with appropriate `LCID` (*e.g.*, `00001`), and sent it to the DCCH channels. Upon receiving this malformed RLC PDU, basebands crash immediately and lose connection with the network. The root cause appears to be a buffer overflow that occurs when the baseband attempts to parse an excessive number of `[E, LI]` pairs.

C.6 B9: Incorrect Handling of 1-byte PDCP Data PDUs for the User Plane

We tested the Galaxy S24 and found that its baseband crashes when receiving a malformed PDCP Data PDU containing just 1 byte. Unlike bug B7 that affects the control plane, this vulnerability impacts the user plane Data PDU structure. The crash likely results from a buffer overread, as user plane Data PDUs require at least 2 bytes for the header. This issue only occurs when the PDCP layer is configured with a 12-bit Sequence Number in State 4.

D Detailed Test Case Generation.

D.1 MAC Layer

Step ①: Generate initial MAC packets. LLFuzz begins by selecting either the DL-SCH or RAR structure to generate seed packets. Since both support multiple sub-headers and sub-payloads, it creates packets with varying numbers of these components (*e.g.*, 1, 3, or 50+). For example, Figure 8 shows a DL-SCH packet with 3 sub-headers. To do

Table 4: Tested devices.

Vendor	No.	Smartphone	Baseband Model	Firmware Ver.	PHY	MAC	RLC	PDCP
Qualcomm	1	SS* Galaxy Note 20 Ultra	Snapdragon 865+	N986NKSU2HWH5	-	B1, B2	B3	-
	2	SS Galaxy S20	Snapdragon 865	G981NKSU3IWH5	-	B1, B2	B3	-
	3	SS Galaxy S22 Plus	Snapdragon 8 Gen 1	S906NKSU2BVJA	-	B2	B3	-
	4	SS Galaxy S24 Ultra	Snapdragon 8 Gen 3	S928NKSU2AXE4	-	-	B3	-
	5	OnePlus 9 Pro	Snapdragon 888	Q_V1_P14	-	B1, B2	B3	-
MediaTek	6	SS Galaxy A31	Helio P65	A315NKOU1DVH1	-	B4, B5, B6	-	-
	7	SS Galaxy A32	Helio G80	A325NKOU4DWH3	-	B4, B5, B6	-	-
	8	Xiaomi K40 Gaming	Dimensity 1200	MOLY.NR15.R3.TC8.PR2.SP.V1.P51	-	-	-	B7
	9	Xiaomi Redmi Note 9T	Dimensity 800U	MOLY.NR15.R3.TC8.PR1.SP.V1.1.P106	-	-	-	B7
SS* Exynos	10	SS Galaxy S21	Exynos 2100	G991NKOU4EWE2	-	-	-	-
	11	SS Galaxy S24	Exynos 2400	S921NKSU2AXE4	-	-	B8	B9
	12	SS Galaxy S10e	Exynos 9820	N970NKOU1ASE5	-	-	-	-
GG† Tensor	13	Pixel 6a	Google Tensor	AP1A.240305.019.A1	-	-	B8	-
	14	Pixel 8 Pro	Google Tensor G3	AP2A.240905.003	-	-	-	-
Huawei	15	Huawei P30 Pro	Kirin 980	21C20B379S000C000	-	-	-	-

∗ Samsung.
† Google.

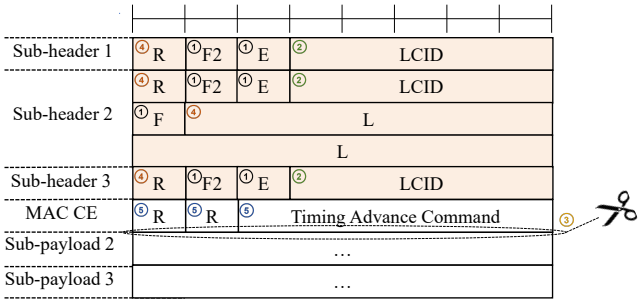


Figure 8: MAC test case generation steps. The numbers correspond to the steps in the generation process.

so, LLFuzz decides the sub-header format based on fields such as `F`, `F2` (DL-SCH), and `T` (RAR). Structural fields (`F`, `F2`, `T`, `E`) are fixed after this step, while others are assigned temporary legitimate values for later mutation.

Step ②: Mutate LCID fields. In the case of the DL-SCH structure, LLFuzz subsequently mutates the `LCID` fields in the sub-headers. LLFuzz combines varying `LCID` values, as they serve as identifiers for logical channels, which may have different priorities in baseband processing. Importantly, the `LCID` value for each sub-header is constrained by its format. For instance, sub-header format A (`F2 = 0`, `F = 0`, `E = 1`) cannot contain `LCID` values reserved for MAC CEs. Using inappropriate `LCID` values could inadvertently alter the sub-header format, disrupting the intended structure.

Step ③: Packet truncation. LLFuzz truncates the generated MAC packets to create various combinations of sub-headers and sub-payloads. For instance, the MAC DL-SCH packet in Figure 8 can be truncated to create packets with: (i) only 3 sub-headers without sub-payloads, (ii) 3 sub-headers and 2 sub-payloads, (iii) 3 sub-headers and 1 sub-payload, or (iv) normal packets with 3 sub-headers and 3 sub-payloads. This step is essential for generating malformed test cases that can potentially trigger memory bugs in the basebands.

Step ④: Header mutation. In this step, LLFuzz mutates the values of the remaining fields (*e.g.*, `RAPID`, `BI`, and `R` in RAR sub-headers, and `L` and `R` in DL-SCH sub-headers). LLFuzz iterates through these fields and mutates them based on their bit lengths. LLFuzz does not mutate all possible values, as this would result in an excessive number of test cases. Instead, it focuses on boundary values and a few representative legitimate values. For example, for the 7-bit `L` field, LLFuzz mutates this field to 0 and $2^7 - 1$.

Step ⑤: Payload mutation. Finally, LLFuzz mutates the fields in the sub-payloads. For the MAC DL-SCH structure, it only mutates the sub-payloads corresponding to MAC CEs. For the RAR sub-payloads, LLFuzz mutates the fields related to RAR grants, such as `Timing Advance Command`, `UL Grant`, `R`, and `Temporary C-RNTI`. All mutations are specification-guided to ensure values remain within valid ranges and do not corrupt adjacent fields.

D.2 RLC Layer

Step ①: Generate initial RLC packets. LLFuzz selects a specific RLC structure from the specification and generates valid RLC packets. Since many RLC formats can include multiple data chunks, LLFuzz creates packets with varying numbers of chunks (*e.g.*, 0, 3, or more than 50). At the end of this step, certain fields in the fixed header (*e.g.*, `D/C`, `E`) are set and not mutated further, as they define the overall structure of the packet.

Step ②: Packet truncation. LLFuzz creates both valid and malformed test cases by truncating the generated RLC packets. When multiple data chunks are present, it generates variants with different numbers of chunks (*e.g.*, 3, 2, 1, or 0). LLFuzz also truncates the fixed header or extension part to produce very short packets, such as those with only 1 or 2 bytes. This approach enables the generation of diverse test cases, including those rarely seen in normal communication.

Step ③: Header mutation. After structural fields are fixed, LLFuzz mutates the remaining fields, such as Frame Information (`FI`), Resegmentation Flag (`RF`), Sequence Number (`SN`), and Length Indicator (`LI`). For fields with short lengths (*i.e.*, fewer than 3 bits), LLFuzz exhaustively mutates all possible values. For longer fields, it focuses on boundary values and representative valid values. For example, for the 11-bit `LI` field, LLFuzz mutates it to 0 and $2^{11} - 1$.

Step ④: Mapping to logical channel. Once an RLC test case is generated, it must be delivered to the correct logical channel. LLFuzz wraps the test case in a MAC container with an appropriate sub-header, specifying the Logical Channel ID (`LCID`) and Length (`L`) fields. For example, a 10-byte test case for the DCCH channel is assigned `LCID` = `00001` and `L` = `10`, ensuring the baseband correctly routes and processes the packet.

D.3 PDCP Layer

Step ①: Generate initial PDCP packets. Similar to the MAC and RLC layers, LLFuzz chooses a specific PDCP structure and generates a valid PDCP PDU. It temporarily assigns legitimate values to the fields in the PDCP header. Note that PDCP packets do not include multiple sub-headers, sub-payloads, or data chunks.

Step ②: Packet truncation. LLFuzz truncates the legitimate PDCP PDU to create both valid and malformed test cases. For example, for the PDCP Data PDU for the control plane, it truncates the PDU to 1-5 bytes, which is shorter than the minimum length of 6 bytes.

Step ③: Header mutation. Then, LLFuzz mutates the remaining fields in the PDCP header, such as `R` and `SN`. It applies the same mutation strategy as in the MAC and RLC layers, focusing on boundary values and a few representative legitimate values. For example, for the 12-bit `SN` field, LLFuzz mutates it to $2^i - 1$ for i ranging from 0 to 12.

Step ④: Mapping to logical channel. To deliver PDCP test cases to the intended logical channels in the baseband, LLFuzz encapsulates them step-by-step through the RLC and MAC layers. It first wraps the PDCP PDU in an RLC PDU, selecting the structure based on whether the target is control or data plane. Then, it encapsulates the RLC PDU in a MAC PDU, setting the `LCID` and `L` fields to guide packet routing. For example, when testing DRBs mapped to DTCH, LLFuzz uses an RLC UM Data PDU (as srsENB typically does for DRBs) and sets `LCID` to `3` with the appropriate length.

D.4 PHY Layer (DCI Messages)

LLFuzz treats Downlink Control Indicator (DCI) structures as complete headers for generating test cases. Since the PHY layer automatically pads missing bytes with zeros before transmission, LLFuzz does not apply truncation-based mutations at this layer. The following steps detail the DCI test case generation process:

Step ①: Generate initial DCI messages. Based on the current eNB configuration (*e.g.*, bandwidth, TDD/FDD mode, transmission mode, CA), LLFuzz selects the corresponding DCI format from the specification and determines the bit lengths of configuration-dependent fields. It then populates the DCI message with legitimate field values captured from normal transmissions. During this step, fields that define the structure of the DCI message are fixed. For example, in DCI format 1, LLFuzz sets the first bit to `1` as it distinguishes between formats 0 and 1.

Step ②: Mutate the remaining fields. LLFuzz mutates the remaining fields in the DCI message, prioritizing those related to resource allocation and those containing reserved values (*e.g.*, `Resource Allocation Type 0/1/2` fields, `Precoding Information`). Similar to other layers, LLFuzz consistently follows the specification to select appropriate mutation values, focusing on boundary, abnormal, reserved, and on a few legitimate samples. For example, for the 17-bit Resource Allocation Type 0 field, LLFuzz mutates the field to 0 and $2^i - 1$ for i ranging from 0 to 17. Typically, a single UE should not use the whole bandwidth; thus, extreme values such as $2^{17} - 1$ are considered abnormal.

Step ③: Allocate PDCCH resources for transmission. DCI messages are transmitted to UEs over the Physical Downlink Control Channel (PDCCH), where each message is encoded into several Control Channel Elements (CCEs). LLFuzz uses the original srsENB allocation algorithms to allocate CCEs for the generated DCIs, ensuring UE-specific values for successful decoding by the target device.