Optimizing 5G Protocol Fuzzing: Structure-Aware Techniques and Feedback Integration

Sara Da Canal[1,2], Francesco Mancini[1,2], Luca Mastrobattista[2] and Giuseppe Bianchi[1,2]

[1]*CNIT National Network Assurance and Monitoring (NAM) Lab, Rome, IT*
[2]*University of Rome "Tor Vergata", Rome, IT*

Abstract
Protocol fuzzing is a technique used to assess the security posture of a network by sending malformed or unexpected inputs. However, in a complex context like 5G, it requires optimization to achieve meaningful results, which involve reaching deep network states that depend on specific message sequences, and testing them. By using structure-aware fuzzing and a feedback mechanism, we derive the protocol's structure to guide input generation and apply targeted optimizations. This approach enhances vulnerability detection and ensures comprehensive testing of the network's resilience.

Keywords
5G Core Network, fuzzing, feedback, structure aware

1. Introduction

The security of mobile networks is of fundamental importance, given their role as critical infrastructure supporting communication, commerce, and essential services. With the introduction of 5G, security has become a predominant concern, with security features embedded in the specifications and a strong focus on security assurance and testing. Specifically, various testing methodologies have been explored, with fuzzing emerging as a widely used technique. This testing method, which consists of stimulating a system with random or unexpected inputs, has seen significant advancements lately, with novel approaches improving its effectiveness in identifying vulnerabilities, and it is widely employed across different domains.

For security certification purposes, it is essential that testing be conducted externally, allowing independent certification bodies to evaluate the network. This requirement necessitates a black-box testing approach, where testers have no access to the network's internal code. While fuzzing through the network interface is a viable solution, it is inherently slower than traditional methods, as every test must pass through the communication interface. This added latency underscores the need for optimization to maintain an effective testing campaign.

In this paper, we apply and adapt known optimization techniques to address the unique challenges of fuzzing a 5G Core Network in a remote testing scenario. Specifically, we developed a fuzzing engine employing both structure-aware fuzzing and feedback, adapting these concepts to the specific context of 5G and considering the added difficulty of remote testing. Our approach demonstrates how these techniques can be tailored to improve the efficiency and depth of fuzzing campaigns in this environment.

The paper is organized as follows: Section 2 provides a brief introduction to the 5G Core Network and fuzzing, highlighting the state of the art on these topics. Section 3 explores the motivation behind our work. Section 4 explains structure-aware fuzzing, its application, and possible optimizations. Section 5 outlines a feedback mechanism for our fuzzer, and section 6 describes our testing architecture and the results obtained.

Joint National Conference on Cybersecurity (ITASEC & SERICS 2025), February 03-8, 2025, Bologna, IT
sara.dacanal@cnit.it (S. Da Canal); francesco.mancini@cnit.it (F. Mancini); luca.mastrobattista@students.uniroma2.eu (L. Mastrobattista); giuseppe.bianchi@uniroma2.it (G. Bianchi)
0009-0006-8928-7096 (S. Da Canal); 0000-0002-1924-8706 (F. Mancini); 0000-0001-7277-7423 (G. Bianchi)

2. Background and Related Works

2.1. 5G Core Network

The increasing demand for flexibility in 5G networks has driven a transition from hardware-based components to software-defined network functions, adopting the paradigm of Software-Defined Networking (SDN) and technologies like Network Function Virtualization (NFV) to enable this softwarization of the network. Thanks to these technologies, a 5G Core is composed of numerous Network Functions (NFs), each responsible for a specific functionality. For instance, the Access and Mobility Management Function (AMF), which we focus on, manages user access to the network and handovers between cells. Interactions between different network components are defined in the 3GPP technical specifications [1]: NFs utilize a service-based interface employing the HTTP protocol to handle internal control plane functionalities, while communication between the user and control plane, or between NFs and other network components, is based on point-to-point protocols, usually developed specifically for cellular communication. For example, our work focuses on the NAS protocol, which governs signaling communications between the UE and AMF.

2.2. Fuzzing

Fuzzing is a widely used testing technique in both software and protocol domains. It consists of sending a large number of unexpected or malformed inputs to a system and analyzing its reactions [2]. The aim is to either cause a crash, uncovering possible buffer overflows, or to discover unexpected behaviors that could stem from vulnerabilities in the system's implementation.

Over time, fuzzing has become increasingly popular and sophisticated, leading to the development of various tools and the standardization of different approaches to this testing method. In the software domain, research has primarily focused on *coverage-guided fuzzing*, which employs code instrumentation to gather coverage information and use it as feedback during the fuzzer's execution [3]. Some works have applied feedback mechanisms in a network environment, such as [4] and [5], but these approaches still require access to source code. On the protocol side, alternative optimizations have been explored, particularly in the realm of *structure-aware fuzzing*, which focuses on crafting fuzzed yet protocol-compliant inputs to maximize the number of meaningful tests. Examples include [6] and [7].

However, applying generic network fuzzers to mobile networks presents several additional challenges and does not always yield the desired results [8]. Some efforts have been made to fuzz LTE networks, with relevant studies including [9, 10, 11]. Regarding the fuzzing of 5G networks, there are works that focus on fuzzing UEs and gNBs, even at lower layers [12, 13], as well as 5Greplay [14], which focuses on testing the Core Network by replaying network packets with and without modifications from pcap traces, though it does not allow for in-depth packet editing.

3. Motivation

In our testing scenario, tests must be conducted in a black-box manner to enable external entities, independent of the network vendor, to perform the testing. However, this setup requires sending messages over the network interface, as it is assumed that the tester does not have direct access to the network under test or its source code, which results in substantial latency between tests. Furthermore, because we cannot isolate specific code areas and must reset the system state after each test, communication needs to be restarted for every attempt, leading to a considerable waste of time. These constraints prevent us from using the high-volume random input approach common in software fuzzing, where testing cycles are much faster. Instead, it becomes essential to design tests that are both targeted and meaningful.

One of the main challenges we face is testing deeper layers within the network. Randomly modified inputs allow us to test the decoding function, but only a minimal percentage of these fuzzed inputs will penetrate further into the system. Instead, our objective is to assess how the system interprets and handles message content, which requires semantically meaningful messages capable of successfully completing the decoding phase. Additionally, the protocol functions as a state machine, where each incoming message triggers a transition from one state to another. This structure means that initial states are relatively easy to test, while deeper states—those that require a specific sequence of messages to reach—are likely to remain untested in most scenarios.

For these reasons, it is essential to optimize which inputs are sent, moving away from a fully random approach and instead drawing inspiration from techniques such as structure-aware fuzzing and guided fuzzing.

4. Preliminary Optimizations: Structure-Aware Fuzzing

Structure-aware fuzzing means that when a system requires complex, structured inputs, the fuzzing process is performed with an understanding of the input's specific structure rather than in a completely random way. This approach requires both preliminary knowledge of the input's semantics and a fuzzing engine capable of making modifications according to those rules. Unlike protocols with ASN.1 specifications [15], which are common in the telecommunications landscape and allow for straightforward automatic parsing, NAS messages are described in tabular format [16] and lack standardized, automatically parsable specifications. Therefore, our initial step was to derive the structure of possible NAS messages to enable a more targeted and effective fuzzing approach. Once these structures are defined, we can specify message modifications through configuration files, which indicate the parts of each message type to alter.

4.1. Messages' Structures Extraction

Representation. While ASN.1 is a common language for specifying network messages in a standardized, independent manner, its strength lies in its conciseness, allowing messages to travel quickly without wasting network resources. However, although we aim to describe a network protocol, our primary need is a representation format specifically for fuzzing inputs, so brevity is not a priority. Instead, we chose to represent inputs as JSON objects, drawing inspiration from fuzzers used to test REST APIs, such as [17]. These fuzzers perform structure-aware fuzzing based on OpenAPI specifications, which provide a standardized, language-agnostic interface to HTTP APIs. This interface allows both humans and machines to understand the service's capabilities and can be represented through YAML or JSON files [18]. JSON objects are easily processed by computers, offer a versatile representation, and maintain human readability.

Messages' structure. NAS messages can be either plain or security-protected: each security-protected message encapsulates an (eventually ciphered) plain-text message. A fundamental principle of our fuzzing campaign is that messages must successfully pass decoding, which requires having messages with a valid MAC. For this reason, modifications are performed on plain-text messages, and the MAC is recomputed afterward if necessary. Another constraint on the structures we can obtain is the scope of our fuzzing campaign: our target is the AMF, but NAS messages can be processed by different NFs, such as the SMF. Nevertheless, we focused on obtaining structures only for Mobility Management NAS messages, although the tool can be expanded.

Each NAS message is composed of a header and a sequence of Information Elements (IEs). The header structure is quite simple and is the same for every message, but it is still included in every file for clarity. It consists of four values: extended protocol discriminator (EPD), some spare bits, security header type, and message type. However, the only field that varies among our messages is the message type field.

IEs are more varied and complex: each message type can have both mandatory and optional IEs, and each IE consists of a combination of type, length, and value. The type field contains the IE identifier (IEI), is present only for optional IEs, and is relative to the specific IE inside the message. Depending on the specific IE, the value can have either a fixed or variable length, and the length field is included only when the value can be variable [19].

This kind of structure is centered around the message. Some applications could benefit from a representation where the focal point is the Information Element, listing the messages to which each IE belongs. We decided to create and store both structures so that the fuzzing engine can use the most suitable one.

Information extraction. The structure of NAS messages is described in the specifications defined by the 3rd Generation Partnership Project (3GPP) [1]. These specifications are written in natural language by hand and contain thousands of pages. Therefore, understanding them comprehensively is nearly impossible, and manual parsing is a cumbersome task. Even automatic parsing requires significant effort due to the lack of a defined structure and the large file size.

For these reasons, we sought a more efficient approach. Some libraries already handle the automatic parsing and generation of NAS messages, such as Pycrate [20], a Python library for manipulating various digital formats related in one way or another to cellular network signaling. This library features classes that describe the possible NAS messages: within each class, the list of all IEs that are usable by the message it represents—including optional ones—is encoded. By employing reflection—i.e., a feature that allows a programming language like Python to examine or "introspect" upon itself—we were able to extract the structure of these classes from Pycrate and convert them into our desired JSON representation.

4.2. Fuzzed inputs generation

Starting from message specifications, we need to derive new inputs for the Core Network. Typically, fuzzing techniques for input generation can be divided into two main categories: generation-based and mutation-based. The former starts with an input grammar to create entirely new inputs, while the latter focuses on mutating existing, valid inputs to generate new ones [21]. We opted for a mutation-based approach, as it ensures that unmodified IEs have valid values, which would otherwise be difficult to generate. To achieve this, we must determine both the types of modifications that can be applied—ensuring they are random but still respect the message's structure to some extent—and how this information can be provided to the fuzzing engine.

Structured modifications applicable to NAS messages can be divided into two categories: modifications affecting the entire information element (IE) and those targeting values within an IE. The former includes actions like adding new IEs, even if the Core Network is not in the appropriate state to process them, while the latter includes creating inconsistencies, such as mismatches between a value and its declared length, which can be more effective in uncovering vulnerabilities like buffer overflows or crashes. Although our testing architecture can monitor the Core Network to detect crashes, it is not designed to identify protocol errors. Nevertheless, we have designed the input generation process to be more generic, enabling its use within other systems.

For modifications involving the entire IE, we implemented a system capable of removing IEs from a message, adding new IEs in various ways—such as inserting an optional IE between mandatory ones—taking an IE from one message and adding it to another, duplicating IEs, or swapping the order of IEs. Intra-IE modifications can be performed either consistently, where the value is changed while maintaining a valid format, or inconsistently, such as creating an IE with a value but no length or adding an IEI to a mandatory IE.

Before each fuzzing run, a JSON file specifying which modifications should be applied to which messages is generated and provided to the fuzzing engine. The engine parses this file and applies the modifications when the corresponding message is intercepted. The structure of this file is simple enough to be written manually, making it a programmable tool. For example, if a specific Core Network is known to crash upon receiving certain messages, those messages can be reproduced directly to replicate the failure. This programmability provides additional advantages, such as greater control over the tests conducted and the ability to bring the network into specific states.

However, fuzzing inherently relies on executing a large number of varied tests, and manually generating such tests is both complex and time-consuming. To address this, we devised a mechanism for

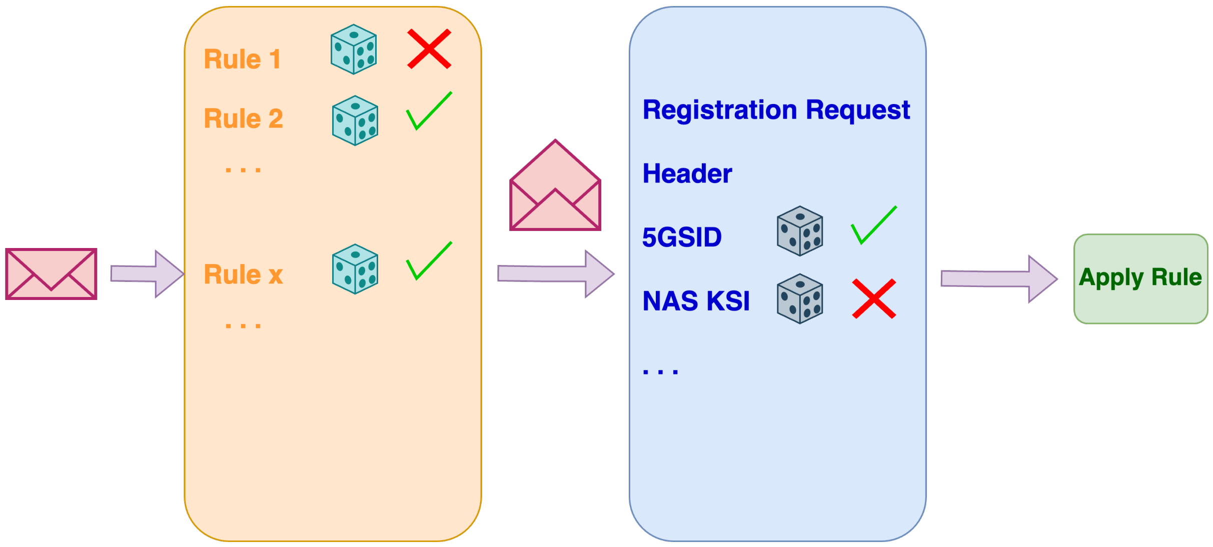

Figure 1: Fuzzed Message Generation

automatically generating a meaningful set of tests using a probabilistic approach based on thresholds. For each fuzzing technique, we set a probability of application, and during input generation, a random number is drawn and compared against this threshold to decide whether the specific rule should be applied. Additional thresholds determine whether a specific IE should be considered for modification, and further thresholds can be incorporated depending on the type of rule being applied. Figure 1 illustrates how the selection process for modifications is performed.

To retain both the benefits of programmability and the higher speed of automatic input generation, we devised a hybrid approach: even when automation is used, a programmer can manually specify certain modifications to apply. During automatic generation, manually specified rules take precedence, and manually modified IEs are never considered for random fuzzing techniques.

Two key points must be highlighted for the correctness of the automatic approach: how the thresholds are chosen and how random numbers are generated. For the former, an initial tuning phase is performed, with details provided in section 4.3, while for the latter, we employed a Lehmer Generator [22].

Pseudo-Random Number Generator. There are several factors to consider when choosing a suitable source of randomness for input generation. First, test repeatability is indispensable: once an interesting result is obtained, the test must be repeatable to pinpoint the bug and verify whether it has been successfully resolved. Pseudo-random number generators produce numbers based on a predetermined sequence defined by an initial seed. This seed can either be manually selected or derived from a truly random value on the machine and then saved for reuse, ensuring repeatability.

Another important consideration is the independence of different extractions. For each modification to an IE, we select multiple values. However, if these values are drawn from the same random stream, correlations may arise, causing some outcomes to have a higher probability than others. This could skew the testing process and reduce its effectiveness. To avoid this problem, we chose to use separate streams based on where the generated value will be used. This is possible because the selected generator offers the functionality of employing multiple independent streams, thereby ensuring statistical independence.

4.3. Tuning

Why? The success of a fuzzing campaign heavily depends on how inputs are generated. Therefore, setting appropriate thresholds to determine which modifications will be applied is essential. These thresholds depend on both the type of rule and the information element involved. For instance, removing an IE has a predictable outcome, whereas the result of modifying a value depends on the new value chosen. To ensure even coverage, the first rule should have a higher threshold. Additionally, thresholds must align with the objectives of the testing campaign.

The goal of our fuzzer is to explore deeper states within the analyzed procedure rather than focusing on inputs that trigger decoding errors because, while such errors are useful for testing decoding functionality, they fail to reach or test other critical aspects of the system. Therefore, thresholds should discourage modifications that create an undecodable message.

How? To perform tuning successfully, we need a deeper understanding of the events that follow each message, as the output obtainable without direct access to the Core Network is insufficient. We opted for a grey-box fuzzing approach for this step: while we do not directly modify the Core Network's software, we assume that logs from the network function under test are accessible. This does not hinder the fuzzer's functionality, as tuning can be performed on an open-source Core Network, with the obtained values later reused on another Core.

Tuning is conducted using the same fuzzing engine employed during testing. We initialize some thresholds, which are incrementally adjusted based on new results. A clearer understanding of the impact of a specific fuzzing rule can be achieved by analyzing each rule in isolation. For this purpose, each strategy is optimized individually, with its initial threshold set to 1, while thresholds for all other techniques are set to 0.

Regarding information elements, assigning a threshold to each individual IE is too cumbersome. Instead, we group them into categories: for each rule, we set different thresholds for mandatory and optional IEs, both initially set to 0.5. During the tuning phase, applied modifications are recorded along with their timestamps. At the end of the run, the Core Network logs are accessed and analyzed, using the timestamps to match the inputs sent to the NF with the resulting errors.

The threshold for each fuzzing strategy is calculated as:

$$threshold = \frac{total\ inputs - inputs\ causing\ decoding\ errors}{total\ inputs}$$

For IEs, the threshold is determined as:

$$threshold = \frac{successful\ inputs\ for\ method\ i}{\sum_j successful\ inputs\ for\ method\ j}$$

5. Runtime Optimizations: Feedback

While tuning provides a useful baseline, it is insufficient for an effective fuzzing campaign, particularly since thresholds are often determined heuristically and may be calculated on a Core Network different from the one under test. Although tuning offers a convenient starting point, runtime optimizations are more adaptable and enable greater efficiency during testing.

Our primary focus when determining thresholds was to minimize decoding errors. However, another critical principle for selecting suitable inputs is recognizing that bugs can be complex and may arise from multiple contributing factors. Reaching such bugs often requires a sequence of fuzzed messages. Given our interest in uncovering complex issues, we implemented a feedback mechanism that prioritizes messages introducing unexpected values while still allowing communication between the UE and the AMF to continue. This approach facilitates deeper exploration of the network function's internal states.

Why? The idea behind feedback is that if a specific modification sequence produces a communication failure, repeating the same sequence on the same Core Network is redundant. The outcome will remain unchanged, providing no additional insights. Conversely, if communication between the UE and AMF persists despite the modification, replaying the same sequence up to that point and continuing to fuzz with different values can be beneficial. Changes in subsequent messages might reveal new issues, as illustrated in Figure 2.

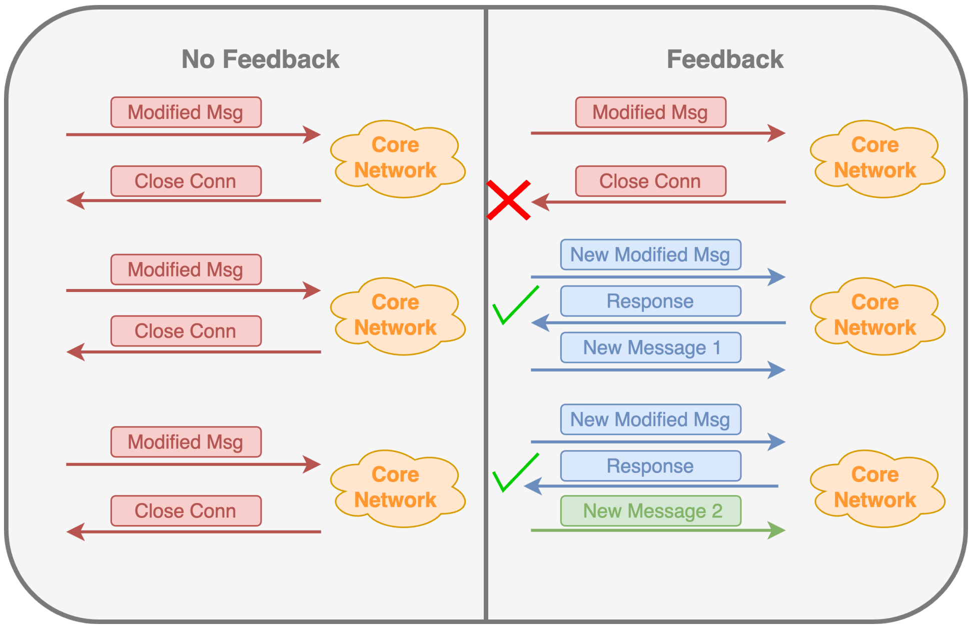

Figure 2: Fuzzing With and Without Feedback

How? Although our fuzzer includes the capability to execute a single test—an especially useful feature for replaying bugs—the general fuzzing approach involves executing a series of registration and deregistration procedures, effectively fuzzing extended message traces. Specifically, at the start of each communication attempt, a fuzzing file is generated and utilized throughout a single fuzzing era. An era ends when a certain amount of time has elapsed, all modifications specified in the file have been tested and discarded, or a configurable number of successful communication attempts—i.e., interface setup—has been achieved. Upon completing an era, a new fuzzing file is created, initiating the next phase of testing.

During each fuzzing era, multiple attempts are made to successfully register and establish a PDU session. When feedback mechanisms are introduced, the fuzzing file is dynamically updated at the end of each registration attempt, ensuring that subsequent attempts incorporate a new set of modifications. This iterative process leads to diverse outcomes and deeper exploration of potential issues.

Initially, we implemented a feedback technique that removes all modifications applied to a message from the fuzzing file whenever the message causes communication between the Core Network and UE to interrupt. While this approach effectively eliminates the blocking modification, it proves overly drastic when a large number of modifications are applied to the same message. Removing all modifications can result in the loss of potentially useful inputs. Additionally, although this feedback technique is valuable for testing deep network states -since eliminating modifications on earlier messages ensures progression to deeper ones- it fails to capture the effects of modification stacking, which may uncover more complex bugs.

A slower but more effective feedback technique involves selectively removing a random subset of modifications applied to a blocking message. If communication blocks again, further subsets of modifications are removed iteratively. This approach increases the likelihood that the resulting messages are processed, as the randomness diminishes with each iteration. However, it still retains some fuzzed inputs while working toward reaching the Core Network's internal states, thereby maintaining the potential to uncover bugs. Although this strategy is slower, it avoids the unproductive focus on unprocessable messages that often characterizes fuzzing without feedback, while preserving the benefits of modification stacking.

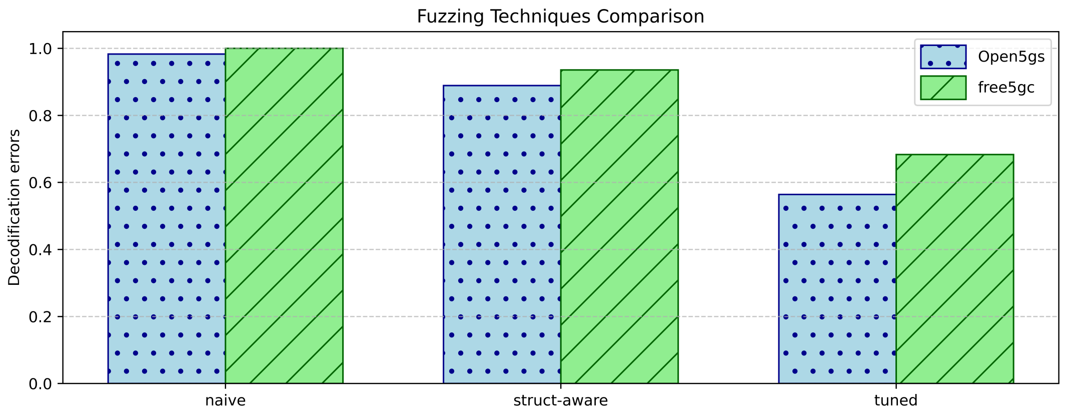

Figure 3: Number of decoding errors encountered with different fuzzing techniques.

6. Testing Architecture and Results

6.1. Testing Architecture

This work builds upon AMFuzz [23], a previous study on fuzzing the 5G Core Network, which focused more on creating a suitable architecture than on optimizing the fuzzing engine. AMFuzz's architecture is based on a proxy that intercepts messages between the gNB and the Core, along with configurable hooks that process intercepted messages, enabling decoding and security handling. The new fuzzing engine has been developed as a series of new hooks, replacing the previous rudimentary approach with a more structured fuzzing mechanism.

Tests can be performed on any Core Network as long as the N2 interface is exposed and configured to allow connections from our framework. Theoretically, multiple instances of the framework could connect to the same AMF, but running parallel tests is ineffective since the results could interfere with each other.

Our tests were conducted on two open-source Core Networks: Open5GS [24] and free5GC [25]. Both Core Networks were deployed using Docker and follow a disaggregated architecture.

6.2. Results

We evaluated our framework based on two key questions:

Q1 Does the tuning process reduce the number of decoding errors encountered during a fuzzing run?

Q2 Does the feedback system enable a deeper analysis of the Core Network?

Structure-aware. Question **Q1** assesses the benefits of introducing structure-aware fuzzing, particularly the impact of tuning compared to applying a raw version of this method. We evaluated our fuzzer by comparing the number of decoding errors caused by each technique. To determine the number of decoding errors, we analyzed the Core Network's logs after executing each method.

To demonstrate that tuning remains effective even when derived from a different Core Network, we generated thresholds using Open5GS and tested both cores with them. As shown in Figure 3, the naive fuzzing engine, which flips random bytes without considering message structure, is never able to produce decodable messages. In contrast, introducing structure-aware fuzzing yields significantly better results. With the addition of tuning, the number of decoding errors decreases even further. As expected, the drop is more pronounced with Open5GS; however, tuning remains portable between different Core Networks.

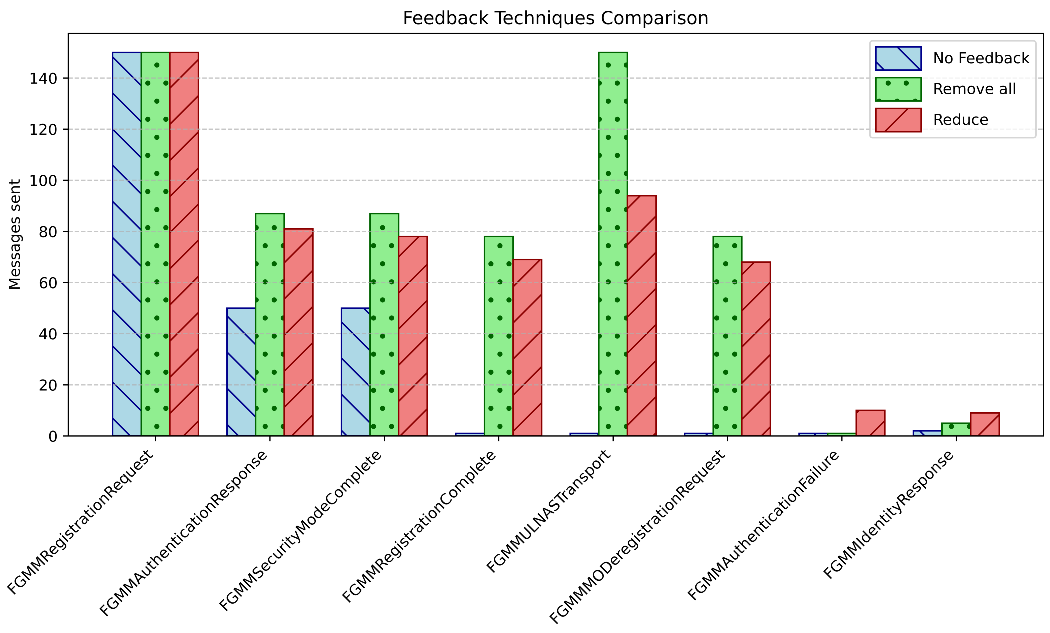

Figure 4: Messages reached with different feedback methods.

Feedback. Question **Q2** evaluates whether introducing a feedback method enables deeper exploration within the Core Network. We tested our feedback techniques on Open5GS by executing different fuzzing runs and measuring how many times a message was reached without feedback or with each of the proposed feedback methods.

As shown in Figure 4, removing all modifications whenever communication stops results in the most even distribution and the highest number of successful registrations. On the other hand, reducing the number of modifications leads to a higher number of deviations from the standard path, as indicated by an increased occurrence of Authentication Failure and Identity Response messages.

Overall, the best feedback method depends on the objectives of the fuzzing campaign. However, both techniques are effective in achieving their respective goals.

7. Conclusions

Both methods proposed for optimizing the fuzzing of a 5G Core Network have proven effective in enhancing the fuzzer's performance, enabling deeper testing of the Core's internal states. Fuzzing, by nature, is not a fast testing technique. Therefore, despite optimizations, achieving meaningful results still requires substantial execution time. However, when the goal is to test the deeper states of the Core Network, such optimizations are essential to prevent an exponential increase in execution time.

Another important outcome of this work is the structured representation of NAS messages, which are easily parsable and hold significant potential for reuse. These structures could be leveraged with alternative feedback or tuning techniques, broadening their applicability. Furthermore, they could serve as a foundation for testing other components, such as the User Equipment, extending the impact of this research.

Acknowledgments

This work was partially supported by the European Union - Next Generation EU under the Italian National Recovery and Resilience Plan (NRRP), Mission 4, Component 2, Investment 1.3, CUP F83C22001690001 and E83C22004640001, partnership on "Telecommunications of the Future"

(PE00000001 - program “RESTART”), and CUP F83C22001670001, partnership on "SEcurity and RIghts In the CyberSpace" (PE00000014 - program "SERICS"). It was also partially supported by the Digital Europe Programme EDIH I-NEST “Italian National hub Enabling and Enhancing networked applications & Services for digitally Transforming SMEs and Public Administrations” G.A. 101083398.s

Declaration on Generative AI

During the preparation of this work, the authors used GPT-3.5 in order to: grammar and spelling check. After using these tools, the authors reviewed and edited the content as needed and takes full responsibility for the publication’s content.

References

[1] 3rd Generation Partnership Project (3GPP), 3gpp portal, https://www.3gpp.org, 1998. Accessed: 2024-10-11.

[2] H. Liang, X. Pei, X. Jia, W. Shen, J. Zhang, Fuzzing: State of the art, IEEE Transactions on Reliability 67 (2018) 1199–1218. doi:10.1109/TR.2018.2834476.

[3] A. Fioraldi, D. Maier, H. Eißfeldt, M. Heuse, AFL++: Combining incremental steps of fuzzing research, in: 14th USENIX Workshop on Offensive Technologies (WOOT 20), USENIX Association, 2020.

[4] V. Pham, M. Böhme, A. Roychoudhury, Aflnet: A greybox fuzzer for network protocols, in: Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track, 2020.

[5] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, T. Holz, Nyx-net: Network fuzzing with incremental snapshots, in: Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys ’22, 2022.

[6] L.-A. Daniel, E. Poll, J. de Ruiter, Inferring openvpn state machines using protocol state fuzzing, in: 2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), 2018, pp. 11–19. doi:10.1109/EuroSPW.2018.00009.

[7] M. Ren, H. Zhang, X. Ren, J. Ming, Y. Lei, Intelligent zigbee protocol fuzzing via&nbsp;constraint-field dependency inference, in: Computer Security – ESORICS 2023: 28th European Symposium on Research in Computer Security, The Hague, The Netherlands, September 25–29, 2023, Proceedings, Part II, Springer-Verlag, Berlin, Heidelberg, 2024, p. 467–486. URL: https://doi.org/10.1007/978-3-031-51476-0_23. doi:10.1007/978-3-031-51476-0_23.

[8] M. Tedman, The challenges of fuzzing 5g protocols, 2021. URL: https://research.nccgroup.com/2021/10/11/the-challenges-of-fuzzing-5g-protocols/.

[9] H. Kim, J. Lee, E. Lee, Y. Kim, Touching the untouchables: Dynamic security analysis of the lte control plane, in: 2019 IEEE Symposium on Security and Privacy (SP), 2019, pp. 1153–1168.

[10] C. Park, S. Bae, B. Oh, J. Lee, E. Lee, I. Yun, Y. Kim, DoLTEst: In-depth downlink negative testing framework for LTE devices, in: 31st USENIX Security Symposium (USENIX Security 22), USENIX Association, Boston, MA, 2022, pp. 1325–1342.

[11] W. Johansson, M. Svensson, U. E. Larson, M. Almgren, V. Gulisano, T-fuzz: Model-based fuzzing for robustness testing of telecommunication protocols, in: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, 2014, pp. 323–332. doi:10.1109/ICST.2014.45.

[12] S. Potnuru, P. K. Nakarmi, Berserker: Asn.1-based fuzzing of radio resource control protocol for 4g and 5g, in: 2021 17th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), 2021, pp. 295–300. doi:10.1109/WiMob52687.2021.9606317.

[13] M. E. Garbelini, Z. Shang, S. Chattopadhyay, S. Sun, E. Kurniawan, Towards automated fuzzing of 4g/5g protocol implementations over the air, in: GLOBECOM 2022 - 2022 IEEE Global Communications Conference, 2022, pp. 86–92. doi:10.1109/GLOBECOM48099.2022.10001673.

[14] Z. Salazar, H. N. Nguyen, W. Mallouli, A. R. Cavalli, E. M. de Oca, 5greplay: a 5g network traffic

fuzzer - application to attack injection, Proceedings of the 16th International Conference on Availability, Reliability and Security (2021).
[15] OSS Nokalva, Inc., Asn.1 made simple: Introduction to asn.1, 2024. URL: https://www.oss.com/asn1/resources/asn1-made-simple/introduction.html, accessed: 2024-11-11.
[16] 3GPP, Non-Access-Stratum (NAS) protocol for 5G System (5GS); Stage 3, Technical Specification (TS) 24.501, 3rd Generation Partnership Project (3GPP), 2023. Version 18.4.0.
[17] V. Atlidakis, P. Godefroid, M. Polishchuk, Restler: Stateful rest api fuzzing, in: Proceedings of the 41st International Conference on Software Engineering, ICSE'19, IEEE Press, Piscataway, NJ, USA, 2019, pp. 748–758. URL: https://doi.org/10.1109/ICSE.2019.00083. doi:10.1109/ICSE.2019.00083.
[18] Swagger, Openapi specification, 2024. URL: https://swagger.io/specification/, accessed: 2024-11-11.
[19] 3GPP, TS 24.007: Mobile Radio Interface Signalling Layer 3, Technical Specification Release 17, 3rd Generation Partnership Project (3GPP), 2022. Available online: https://www.3gpp.org/.
[20] P1Sec, Pycrate, 2016. URL: https://github.com/pycrate-org/pycrate, repo: https://github.com/pycrate-org/pycrate.
[21] J. Fell, A review of fuzzing tools and methods, PenTest Magazine (2017).
[22] S. K. Park, K. W. Miller, Random number generators: good ones are hard to find, Commun. ACM 31 (1988) 1192–1201. URL: https://doi.org/10.1145/63039.63042. doi:10.1145/63039.63042.
[23] F. Mancini, S. Da Canal, G. Bianchi, AMFuzz: Black-Box Fuzzing of 5G Core Networks, in: 2024 19th Wireless On-Demand Network Systems and Services Conference (WONS), IEEE, 2024, pp. 17–24.
[24] Open5GS, Open5gs: Open source implementation for 5g core and epc, 2023. URL: https://open5gs.org/, repo: https://github.com/open5gs/open5gs.
[25] free5GC, free5gc: open-source project for 5th generation (5g) mobile core networks, 2022. URL: https://www.free5gc.org/, repo: https://github.com/free5gc/free5gc.