2021 17th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob) | 978-1-6654-2854-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/WiMob52687.2021.9606317

Berserker: ASN.1-based Fuzzing of Radio Resource Control Protocol for 4G and 5G

Srinath Potnuru
Department of Computer Science
KTH Royal Institute of Technology, Sweden
potnuru@kth.se

Prajwol Kumar Nakarmi
Business Area Networks
Ericsson, Sweden
prajwol.kumar.nakarmi@ericsson.com

***Abstract*—Telecom networks together with mobile phones must be rigorously tested for robustness against vulnerabilities in order to guarantee availability. RRC protocol is responsible for the management of radio resources and is among the most important telecom protocols whose extensive testing is warranted. To that end, we present a novel RRC fuzzer, called *Berserker*, for 4G and 5G. Berserker's novelty comes from being backward and forward compatible to any version of 4G and 5G RRC technical specifications. It is based on RRC message format definitions in ASN.1 and additionally covers fuzz testing of another protocol, called NAS, tunneled in RRC. Berserker uses concrete implementations of telecom protocol stack and is unaffected by lower layer protocol handlings like encryption and segmentation. It is also capable of evading size and type constraints in RRC message format definitions. Berserker discovered two previously unknown serious vulnerabilities in srsLTE – one of which also affects openLTE – confirming its applicability to telecom robustness.**

***Keywords*—Fuzzing, security, RRC, NAS, ASN.1, 4G, 5G**

I. Introduction

Context. Telecom networks enable not only mobile broadband connection, but also (increasingly) industry automation and critical communications. They are required to have 99.999% (five 9s) availability, which is less than six minutes of downtime in a year [1]. To ensure high availability, telecom networks must be rigorously tested for robustness against failures.

Radio Resource Control (RRC) is a Layer 3 protocol used for radio resource management between mobile phones and a base station. The RRC protocol also tunnels an upper layer protocol called Non-Access Stratum (NAS). Without graceful handling of RRC messages, there would not be a communication channel. Therefore, it is imperative that the RRC protocol (including the tunneled NAS protocol) is handled in a robust manner by both the mobile phones and the network.

To ensure robustness, manufacturers need to perform two broad categories of software testing, i.e., conformance and fuzz testing. While conformance testing ensures that a System Under Test (SUT) works properly with expected messages, fuzz testing tries to identify vulnerabilities in the SUT by sending unexpected messages [2]. Fuzz testing complements conformance testing by discovering faults not identified by the latter, e.g., buffer overflow and race conditions. Discovery of faults contributes to a robust software by giving the manufacturers an opportunity to fix them before the software is shipped. Hence, fuzz testing of the RRC and the tunneled NAS protocols is indispensable to ensure robust telecom networks.

Related work. BASESPEC [3] proposes Layer 3 testing by comparing baseband firmware to cellular specification, however, such comparison is mainly limited to conformance testing and not to fuzzing. BaseSAFE [4] fuzzes a mobile phone baseband by collecting downlink messages; using AFL++ [5] on those messages to generate a fuzzing corpus; and replaying the corpus to the mobile phone baseband. LEFT [6] fuzzes mobile phones but by perturbing the order of messages in a modified concrete implementation of a base station. Works such as LTEInspector [7], LTEFuzz [8], and 5GReasoner [9] generate and use test cases based on manual analysis of 3GPP technical specifications (TS). A major hurdle with fuzzing the RRC protocol using the above-mentioned telecom fuzzers is that they are tied to the implementation of a particular 3GPP TS version. It makes the fuzzers non-backward compatible with older SUTs and soon-to-be obsolete with newer SUTs, because 3GPP TSes are continuously evolving for new features and enhancements.

There are fuzzers like T-Fuzz [10] for NAS and T3FAH [11] for SIP that are based on TTCN-3 schema and do not natively embed message formats. But they too are not sustainable for fuzzing the RRC protocol since RRC messages are not defined in 3GPP TSes using TTCN-3 format.

Our work. Despite its importance – to our knowledge – we see a research gap in fuzz testing of the RRC protocol that can cope with continuously evolving 3GPP TSes. To fill this research gap, we present the design and evaluation of our backward and forward compatible RRC fuzzer for 4G and 5G – called *Berserker*[1].

Berserker relies on a concrete implementation of telecom protocol stack and can fuzz test both the network side (uplink) and the mobile phone side (downlink) as SUTs (§II). Berserker acts as an additional **shim** layer in the concrete implementations and mutates or replaces RRC messages before they are sent to lower layers and ultimately to the SUT. Thus, Berserker is unaffected by lower layer protocol handlings like encryption, integrity protection, segmentation, and scheduling.

[1]Berserkers are historic Nordic warriors known to be extremely furious; the name represents our fuzzer that is ruthless with a system under test.

We designed Berserker to obtain knowledge of RRC messages from whichever version of 3GPP TS 36.331 [12] for 4G and TS 38.331 [13] for 5G it is provided with. It extracts RRC Abstract Syntax Notation One (ASN.1) [14] schema definitions directly from a 3GPP TS and compiles the extracted schema to produce a corresponding encoder and decoder.

However, encoder and decoder based on the RRC ASN.1 schema follow constraints imposed by the ASN.1 schema on what values a field can take; e.g., if a field is defined to be 40-bits long, then the encoder cannot set that field to 80-bits long value. To overcome this, we added a capability in Berserker to mutate the RRC ASN.1 schema itself, e.g., changing the field definition from 40-bits to 80-bits.

We evaluated Berserker with the open-source project, srsLTE [15], in two configurations (§III): (a) srsUE as the concrete implementation of 4G mobile phone; srsENB and srsEPC as the 4G network SUT, and (b) srsENB and srsEPC as the concrete implementation of 4G network; srsUE as the 4G mobile phone SUT. Berserker discovered two serious vulnerabilities in the srsEPC, thus proving its effectiveness (§IV). Upon further investigation, we learned that one of them is inherited from another open-source project, openLTE [16].

To summarize, our contributions are:

1) We designed a novel RRC fuzzer, *Berserker*, that is both backward and forward compatible with continuously evolving 3GPP TSes by extracting RRC ASN.1 schema directly from any 3GPP RRC TS for 4G and 5G.
2) We present a technique of sidestepping constraints in RRC ASN.1 schema by mutating the schema itself.
3) We evaluate the fuzzer using open-source components in srsLTE and discovered two previously unknown serious vulnerabilities in srsLTE, one of which also affects openLTE.
4) We prove that RRC fuzzer can act a machinery to effectively fuzz upper layer NAS protocol.
5) By using concrete implementations of telecom protocol stack, we show that the RRC (and NAS) layer can be fuzzed while staying unaffected by lower layer protocol handlings.

Responsible disclosure. We have informed the srsLTE team about the crash. For wider disclosure to commercial manufacturers, other relevant open-source projects (including openLTE) as well as their users, we are coordinating with Ericsson's PSIRT.

II. Berserker: RRC fuzzer for 4G and 5G

Berserker comprises of two main components called *Fuzzer* and *Driver* as shown in Fig. 1. In what follows, we give detail description of these components.

A. Berserker Fuzzer

Berserker Fuzzer component is responsible for fuzzing the RRC messages (including the tunneled NAS messages). It supports both: mutating the contents of an existing RRC message – called *mutation-based* fuzzing – and generating a new RRC message from scratch – called *generation-based*

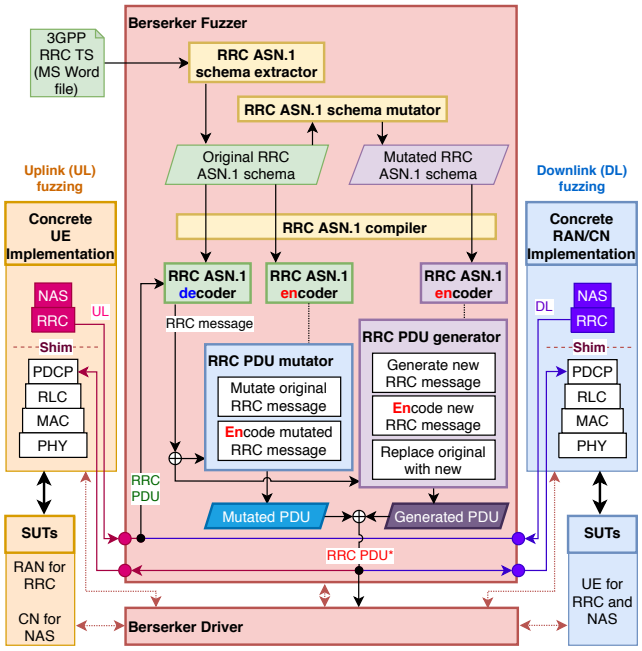

Fig. 1: Components of Berserker. When RAN/CN are the SUTs, only the UE's UL messages are fuzzed while the RAN/CN's UL/DL protocol handlings are untouched. When UE is the SUT, only the RAN/CN's DL messages are fuzzed and the UE's UL/DL protocol handlings are untouched.

fuzzing. When doing mutation-based fuzzing, Berserker conforms to the ASN.1 constraints of the original RRC message. When doing generation-based fuzzing, Berserker replaces the original RRC message with a random RRC message. The replaced message could be of a different type than the original message as well as not conforming to the original ASN.1 constraints.

It implements no part of RRC procedure handling and state management, instead relies on a concrete implementation of UE (for uplink fuzzing) and RAN/CN (for downlink fuzzing) to setup the communication in a state that RRC messages are sent to the SUT. The protocol handling of SUTs (RAN/CN for uplink fuzzing and UE for downlink fuzzing) are untouched.

Berserker operates as below: acts as an additional shim layer in the concrete implementation between the Layer 3 and the lower layers; intercepts RRC messages in Layer 3 before they are passed down to lower layers; fuzzes or replaces the intercepted RRC messages; and passes down the fuzzed or replaced messages to lower layers. It comprises of following sub-components:

RRC ASN.1 schema extractor. This component extracts all RRC message definitions from a 3GPP RRC TS and produces an *original RRC ASN.1 schema*. First, an appropriate version of RRC TS is manually identified based on what is suitable for testing, e.g., which version the concrete implementation of UE or RAN/CN, and the SUTs support. Different versions of RRC TSes for 4G and 5G are publicly available as MS Word files [12], [13]. Next, this component extracts the complete RRC

TABLE I: Mutation strategies for RRC ASN.1 schema

Strategy	**Description** *(examples relate to TS 38.331 [13])*
Change primitive data types	Change type of data that are BOOLEAN, INTEGER, ENUMERATED, OCTET STRING, and BIT STRING. *E.g., change spare from BIT STRING to BOOLEAN.*
Change structured data types	Change type of data that are SEQUENCE, SEQUENCE OF and CHOICE. *E.g., change rrcSetupRequest from RRCSetupRequest-IEs to InitialUE-Identity.*
Extend options	Add new options for data that are ENUMERATED, SEQUENCE and CHOICE. *E.g., add fuzz-code1 to EstablishmentCause.*
Reduce options	Remove existing options for data that are ENUMERATED, SEQUENCE and CHOICE. *E.g., remove randomValue from InitialUE-Identity.*
Scramble options	Change the order of options for data that are SEQUENCE and CHOICE. *E.g., change the order in RRCSetupRequest-IEs to spare, ue-Identity, establishmentCause.*
Change size	Change upper and lower limits of size for data that are INTEGER, OCTET STRING, BIT STRING, and SEQUENCE OF. *E.g., change the size of ng-5G-S-TMSI-Part1 from 39 bits to 99 bits.*

ASN.1 schema into a single plain text file which contains all the text paragraphs between an `ASN1TSTART` tag and the following `ASN1STOP` tag in the order they appear throughout the TS.

RRC ASN.1 schema mutator. It mutates a RRC ASN.1 schema to sidestep the constraints in that schema. The intuition behind sidestepping constraints in the schema is to increase chances of finding bugs in SUTs by sending partially or completely non-conforming RRC messages. This component takes an original RRC ASN.1 schema produced by the RRC ASN.1 schema extractor, performs various mutation strategies (listed in Table I), and produces a *mutated RRC ASN.1 schema*.

RRC ASN.1 compiler. This component takes the original and mutated RRC ASN.1 schema files, which are platform agnostic plain text files, and compiles them to produce programing language specific encoder and decoder of RRC messages.

RRC ASN.1 encoder and decoder. Encoder converts programming language specific RRC message structures into ASN.1 encoded series of bytes, and decoder does the opposite. They conform to constraints in the schema from which they were compiled. Berserker has one encoder and decoder pair compiled from the *original* RRC ASN.1 schema, and another (only) encoder compiled from the *mutated* RRC ASN.1 schema. So, there is one decoder that takes the intercepted RRC Protocol Data Unit (PDU) – ASN.1 encoded series of bytes – from the concrete UE or RAN/CN implementation, and produces a decoded RRC message structure. It is required for the decoder to conform to the original RRC ASN.1 schema because otherwise decoding of the intercepted RRC PDU will fail. There are two encoders. We discuss their use below.

RRC PDU mutator. It is responsible for mutation-based fuzzing in which the contents of an *existing* message are modified. It takes the decoded RRC message from the decoder; performs various mutation strategies listed in Table II; and produces a mutated PDU (called RRC PDU*) using the RRC ASN.1 encoder (compiled using original RRC ASN.1 schema). This RRC PDU* is then forwarded to lower layers. We note that, encoder used by this mutator component conforms to original RRC ASN.1 schema. It is our design choice to focus mutation-based fuzzing within same constraints as of the intercepted RRC PDU. This way, it is guaranteed that the RRC PDU* will never be early rejected by the SUTs (assuming robust SUTs) because of non-conformance.

TABLE II: Mutation and generation strategies for RRC messages

Data type[a]	**Strategy**[b]
BOOLEAN *(primitive)*	Randomly set to True or False.
INTEGER *(primitive)*	Set to random integer within the corresponding SIZE constraints.
ENUMERATED *(primitive)*	Set to a random chosen option from corresponding enum options.
BIT STRING *(primitive)*	If bounded by SIZE constraints, set to a random bit series of random length within the SIZE constraints. Otherwise, if unbounded, set to a random bit series of length 8000[c]. (or use radamsa[d])
OCTET STRING *(primitive)*	If bounded, set to a random octet series of random length within the SIZE constraints. Otherwise, if unbounded, set to a random octet series of length 1000[c]. (or use radamsa[d])
SEQUENCE *(structured)*	For mutation, traverse to find primitive data types, and apply corresponding strategy above when found. For generation, traverse for structured and primitive data types, and initialize them with corresponding strategy above.
SEQUENCE OF *(structured)*	For mutation, traverse to find primitive data types, and apply corresponding strategy above when found. For generation, initialize a random number of corresponding structured or primitive data type with corresponding strategy above.
CHOICE *(structured)*	For mutation, traverse to find primitive data types, and apply corresponding strategy above when found. For generation, randomly choose one of corresponding options and initialize the chosen structured or primitive data type with corresponding strategy above.
OPTIONAL *(qualifier)*	For mutation, remove included field or include missing field. For generation, randomly choose to include field. If included, initialize with corresponding strategy[e] above.
Specific name *(custom, for all data types)*	In mutation, a specific field name can be chosen if multiple fields are of same type. Not used in generation.
Perturbation *(custom, for messages)*	Not used in mutation. In generation, messages are replaced out-of-order.

[a]Working on ASN.1 data types instead of concrete RRC fields makes Berserker generic to any version of 4G and 5G 3GPP RRC TSes.
[b]Strategies for both the mutation- and generation-based fuzzing always conform to the constraints of underlying ASN.1 schema. In generation -based fuzzing, sidestepping the constraints comes from the fact that the underlying schema is mutated, not from non-conforming strategy.
[c]The chosen length is based on our empirical observation with srsLTE that messages above certain length do not get transferred, although up to 8188 octets should have been supported [17].
[d]radamsa is an open-source general-purpose fuzzer [18].
[e]In a so-called Christmas tree message, **all** optional fields are included.

RRC PDU generator. This component does generation-based fuzzing in which a *new* message is generated from

scratch. It instantiates a randomly chosen empty RRC message; fills the content based on various generation strategies listed in Table II; generates a RRC PDU* using the RRC ASN.1 encoder compiled from the mutated RRC ASN.1 schema; and replaces the original RRC message with the RRC PDU*. This RRC PDU* is submitted to lower layers.

This component performs two types of tests on the SUTs. One type is to test how the SUTs handle RRC messages (including tunneled NAS) deviating from constraints in the original RRC ASN.1 schema. In this case, the encoder used by this component conforms to the mutated RRC ASN.1 schema. Another type is to test how the SUTs handle out-of-order messages (also known as perturbation or sequence fuzzing). In this case, we made a design choice to *not* mutate the schema, meaning that the encoder is as good as conforming to the original RRC ASN.1 schema. By doing so, the RRC PDU* will not be early rejected by the SUT on account of non-conformance.

B. Berserker Driver

Berserker Driver component is responsible for monitoring the concrete UE or RAN/CN implementations, and the SUTs; and providing feedback to the Berserker Fuzzer component. It also orchestrates the fuzzing by e.g., controlling seed for randomness; configuring mutation strategies for ASN.1 schema; configuring mutation and generation strategies for RRC messages; triggering transfer of RRC messages; restarting the concrete UE or RAN/CN implementations, and the SUTs if unresponsive.

III. Experiments

Components. In order to evaluate Berserker's design (Fig. 1), we did experiments by using the components listed in Table III. Among them, asn1c [19] and srsLTE [15] are the most prominent. asn1c is an open-source ASN.1 compiler that converts ASN.1 schema into C source code (compatible with C++). srsLTE is an open-source implementation of 4G protocol stack written in C++ and comprising of a CN (called srsEPC), a RAN (called srsENB), and a UE (called srsUE). We chose srsLTE because not only it is popular among telecom security researchers, but also an open-source implementation of 5G protocol stack was unavailable at the time of writing (OAI's [21] 5G RAN and 5G CN are not yet 5G ready). We note that although our experiments are in 4G, they are sufficiently general because the control plane protocol stack is layered in the same order in 4G and 5G, and the RRC protocol in both are defined using ASN.1. For mutation-based fuzzing of BIT STRING and OCTET STRING, we used radamsa [18] in addition to random data.

RRC and NAS messages in our experiments. Berseker covers all messages observed during an initial attach procedure in srsLTE. In uplink, Berserker covers six RRC messages (RRCConnectionRequest, RRCConnectionSetupComplete, ULInformationTransfer, SecurityModeComplete, UECapabilityInformation, and RRCConnectionReconfigurationComplete) and four NAS messages (AttachRequest, AuthenticationResponse, SecurityModeComplete, and AttachComplete). In downlink too, it covers six RRC messages (RRCConnectionSetup, DLInformationTransfer, SecurityModeCommand, UECapabilityEnquiry, RRCConnectionReconfiguration, and RRCConnectionRelease) and four NAS messages (AuthenticationRequest, SecurityModeCommand, AttachAccept, and EMMInformation). When doing mutation-based fuzzing, the RRC messages (including tunneled NAS messages) are modified. When doing generation-based fuzzing, they are replaced with any RRC message (that may include a tunneled NAS message) available in the RRC ASN.1 schema.

TABLE III: Experiment components

Component	Details
SUTs	srsENB/srsEPC (for uplink fuzzing) and srsUE (for downlink fuzzing) applications in srsLTE[a]. srsENB terminates RRC, srsEPC terminates NAS, and srsUE terminates both RRC and NAS. Inactivity timer at srsENB was set to 5 seconds.
Concrete implementations	srsUE (for uplink fuzzing) and srsENB/srsEPC (for downlink fuzzing) applications in srsLTE[b]. They generate RRC and NAS (tunneled in RRC) messages and send them to SUTs.
RF front-end	ZeroMQ-based RF driver[c] provided by srsLTE. It exchanges IQ samples over TCP.
3GPP RRC TS	3GPP TS 36.331 [12] version 15.4.0 (2019-02-19). It was chosen based on what srsLTE supports. It is a MS Word file sized 11.9 MB.
RRC ASN.1 schema extractor	Our inhouse tool[d]. It extracts RRC ASN.1 schema from a 3GPP RRC TS. The extracted schema from the above TS is a plain text file sized 668 KB.
RRC specification mutator	A Python (version 3.8) script. It produces a plain text file with mutated RRC ASN.1 schema. There are multiple mutated files.
RRC ASN.1 compiler	asn1c. We use a fork[e] [19] of original asn1c to simultaneously work with encoders and decoders compiled from different ASN.1 schemas.
RRC encoders and decoder	Namespace-separated custom wrapper around C files produced by the fork of asn1c.
Shim, RRC mutator, and generator	Integrated into `send_<ul,dl>_ccch` and `send_<ul,dl>_dcch` functions in srsUE's `src/stack/rrc/rrc.cc` (for uplink fuzzing) and srsENB's `src/stack/rrc/rrc.cc` (for downlink fuzzing). radamsa[f] is also integrated here.
Driver	Combination of bash scripts and configuration files.
Test machines (virtual)	4 x (4GB RAM, Intel Xeon E3-12xx v2 of x86_64), 1 x (10GB RAM, Intel(R) Core(TM) i5-8350u CPU @ 1.70ghz) all using Ubuntu 18.04 64-bit with gcc version 7.5.0. Each virtual machine had a complete experiment setup.

[a, b] srsLTE – c892ae56be5302eaee5ca00e270efc7a5ce6fbb2 commit tag, Release 20.04.1.
[c] An actual RF front-end and transmission on the air interface is unnecessary for our purpose of fuzzing the RRC protocol since Berserker is unaffected by lower layer protocols.
[d] An alternative publicly available tool is at [20]. Clause A.3.1.1 in [13] describes how one can make their own extractor.
[e] asn1c fork – 27eaf82abed937a2da5a5fd9e0d4076d9abcfc75 commit tag.
[f] radamsa – d71c384bafb53865a561684035fbea1cf6c76734 commit tag.

Collecting results. The Driver continuously triggers RRC messages by either (re)starting the srsUE or by initiating an IP ping request via srsUE. It stores all the mutated and generated RRC PDUs produced by the Fuzzer (PDU* in Fig. 1) and application logs from srsUE, srsENB and srsEPC.

During mutation-based fuzzing, 44,383 uplink and 17,293

TABLE IV: Partial results from mutation-based fuzzing – UECapabilityInformation

Fuzzing strategies	Mutated RRC PDUs	Added time for fuzzing (ms)	Seeds used
INTEGER	67	0.29-2.08	0-10
ENUM	72	0.33-2.03	0-10
OCTET STRING	81	0.23-3.23	0-100
Append to ueCapabilityRAT-Container	6431	0.32-4.14	0-100
Append to ueCapabilityRAT-Container and include optional fields	2905	0.46-5.01	0-100
Fuzz existing and append to ueCapabilityRAT-Container, include optional fields	93	0.41-3.66	0-100
Include optional fields	3098	0.35-4.59	0-100
INTEGER and ENUM	1431	0.34-3.77	0-50
radamsa fuzzing OCTET STRING	133	0.56-5.71	0-100

TABLE V: Partial results from generation-based fuzzing – UECapabilityInformation

ASN.1 specification mutation	Mutation selected	Replaced RRC PDUs	Added time for fuzzing (ms)
Change structured type	UE-CapabilityRAT-ContainerList from *"SEQUENCE (SIZE (0..maxRAT-Capabilities)) OF UE-CapabilityRAT-Container"* to *"UE-CapabilityRAT-Container"*	45	0.61-3.66
	UE-CapabilityRAT-ContainerList to UECapabilityInformation-v8a0-IEs	60	0.64-3.38
Extend options in	RAT-Type: add new-field1, new-field2, new-field3	49	0.49-3.79
	UECapabilityInformation-r8-IEs: add new-field of type UE-CapabilityRAT-ContainerList	54	0.55-8.74
Change size constraints of	RRC-TransactionIdentifier to INTEGER (0..31)	46	0.63-6.13
	ueCapabilityRAT-Container to OCTET STRING (SIZE(20))	46	0.59-3.29
Perturbation[a]	Send other UL-RRC messages in place of UECapabilityInformation	11	0.51-4.30

[a]Uses non-mutated ASN.1 schema while generating other RRC messages. Seeds 0-10 were used for all cases.

downlink RRC messages were mutated. During generation-based fuzzing, 2,598 uplink and 1,997 downlink RRC messages were replaced. Partial results from our experiments are shown in Table IV and V for UECapabilityInformation which is only one of 12 RRC messages. Full results could not be included due to page limitation, but could be made available upon request.

Table IV is for mutation-based fuzzing where each row indicates the strategies used by Berserker's RRC PDU mutator according to Table II. Table V is for generation-based fuzzing where each row identifies the mutations on ASN.1 schema according to Table I; not all mutations are listed; during perturbation, in addition to mutating ASN.1 schemas, Berserker sends other RRC messages that are generated using original ASN.1 schema.

Seeds make the experiments reproducible by serving as initialization values for randomizations; and also, they limit the number of iterations. We manually choose stop value for seeds (also) depending on what is being fuzzed. For example, only 8 iterations are sufficient for an ENUMERATED type with 8 options; and only few hundreds of values are iterated instead of all possible values (2^{40}) for a BIT STRING of SIZE

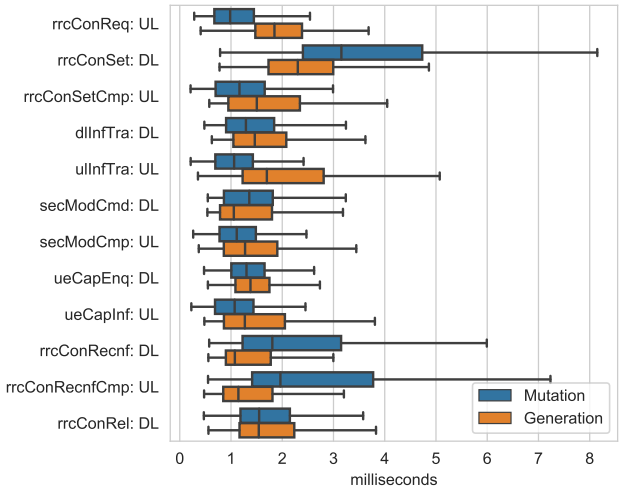

Fig. 2: Box plot of the Fuzzer time cost, i.e., time taken between receiving a RRC PDU and producing a RRC PDU*. Outliers and radamsa are omitted.

40. We also note that the number of fuzzed or replaced RRC messages vary even for the same stop seeds. It is because the srsUE sometimes performs several reconnection attempts itself without trigger from the Driver.

Filtering out false positives. The Driver identifies unexpected behavior, e.g., becoming unresponsive or crashing, of the SUTs through their logs. The srsLTE application logs have a line "*srsLTE crashed... backtrace saved in './srsLTE.backtrace.crash'...*" when the application crashes. But all the unexpected behaviors are not necessarily caused by the Fuzzer; some may be caused by artifacts of the applications themselves. In order to filter out false positives, the Driver has an automated replaying mechanism of RRC PDU* to confirm the unexpected behavior across multiple machines.

IV. Findings

Two vulnerabilities in srsEPC. Berserker found two previously unknown vulnerabilities, both of which cause the srsEPC – the CN part of srsLTE – to crash. One of them was found through mutation-based uplink fuzzing and another through generation-based uplink fuzzing. The root cause of one vulnerability is buffer overflow because of improper parsing of a NAS message. It is triggered by a *fuzzed* RRC message (RRC PDU* from **mutation**-based fuzzing) that tunneled a NAS message. We observed "*stack smashing detected*" error; it originates from safety mechanism against buffer overflow in C++ applications compiled using gcc. The other vulnerability's root cause is invalid memory address access because of improper security processing of a NAS message. It is triggered by a *replaced* RRC message (RRC PDU* from **generation**-based fuzzing) that causes the srsEPC to crash.

Because of the sensitive nature of telecom networks, we do not disclose the exact byte sequences or message names in this paper. But full details could be made available via Ericsson PSIRT.

One vulnerability in openLTE. We investigated further and learned that srsLTE uses some lines of codes from openLTE [16] (another open-source LTE project) for parsing NAS messages. We fed the fuzzed RRC message, that caused buffer overflow in srsLTE, into openLTE (`a5a66ed660e6094a9f50f7e00a0e9805dfbac724` commit tag, 30th Jul 2017 – the latest commit at the time of writing) and discovered that openLTE crashes too. This suggests that the buffer overflow vulnerability in srsLTE came from openLTE's code base.

Effect in srsENB and srsUE. In our experiments, Berserker did not identify any fuzzed RRC message that crashes srsENB or srsUE. Although, during generation-based fuzzing (that uses mutated RRC ASN.1 schema), their application logs contain several errors like "*Invalid field access*", "*Buffer size limit was achieved*", and "*Failed to unpack*". It is yet to be seen if running Berserker for more time (with higher stop values for seeds) will uncover any vulnerability in srsENB and srsUE.

Time cost. The Fuzzer time cost, shown in Fig. 2, is the time taken between receiving a RRC PDU and producing a RRC PDU*. For uplink fuzzing, delay requirements on the UE side are relevant, which are specified in Clause 11.2 of 3GPP TS 36.331 for 4G with minimum requirements up to 10ms. Similar requirements for 5G are defined in Clause 12 of 3GPP TS 38.331, which are more stringent with minimum requirements up to 5ms. As seen in Fig. 2, Berserker mostly introduces 1-2ms of delay for uplink fuzzing which is well within the range of 4G and 5G requirements. In downlink, RAN itself is responsible for scheduling and 3GPP RRC TSes do not specify delay requirements on the RAN side. Nevertheless, as seen in Fig. 2, Berserker introduces 1-3ms of delay for downlink fuzzing, which is in similar range as for uplink.

V. Conclusion and Future Work

Berserker's ASN.1-based design shows good results by discovering previously unknown vulnerabilities in two open-source telecom projects, srsLTE and openLTE. Its main advantage comes from requiring minimal maintenance after initial integration with a system under test, because it is compatible with any version of 4G and 5G 3GPP RRC technical specifications. Berserker can sidestep the constraints on sizes and types imposed by RRC ASN.1 schema and is unaffected by lower layer protocol handlings. It covers fuzzing of not only the RRC protocol but also the NAS protocol tunneled in RRC. In fact, the discovered vulnerabilities are caused by improper parsing and security processing of NAS messages by the network. Discovery of vulnerabilities provides manufacturers of mobile phones and network equipments an opportunity to fix them. Hence, Berserker is very relevant to increasing telecom networks' robustness.

For future work, ability to probe the SUTs' system resources like CPU and memory usage would be a useful addition to Berserker. Berserker also needs to be upgraded for coverage-guided fuzzing and integration with symbolic executions. Further, when Berserker sends out-of-sequence messages in generation-based fuzzing, it does so in random without any knowledge of RRC or NAS protocol states, therefore, Berserker could be enhanced to do smart sequence fuzzing.

References

[1] P. Linder. (2016) How cloud and networks achieve 99.999% availability in different ways. Online. https://www.ericsson.com/en/blog/2016/9/how-cloud-and-networks-achieve-99.999-availability-in-different-ways.

[2] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.

[3] E. Kim, D. Kim, C. Park, I. Yun, and Y. Kim, "BASESPEC: Comparative Analysis of Baseband Software and Cellular Specifications for L3 Protocols," in *Network and Distributed Systems Security (NDSS) Symposium 2021)*, 2021.

[4] D. Maier, L. Seidel, and S. Park, "BaseSAFE: Baseband SAnitized Fuzzing through Emulation," in *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2020, pp. 122–132.

[5] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining Incremental Steps of Fuzzing Research," in *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*, 2020, Online. https://github.com/AFLplusplus/AFLplusplus.

[6] K. Fang and G. Yan, "Emulation-Instrumented Fuzz Testing of 4G/LTE Android Mobile Devices Guided by Reinforcement Learning," in *European Symposium on Research in Computer Security*. Springer, 2018, pp. 20–40.

[7] S. Hussain, O. Chowdhury, S. Mehnaz, and E. Bertino, "LTEInspector: A Systematic Approach for Adversarial Testing of 4G LTE," in *Network and Distributed Systems Security (NDSS) Symposium 2018*, 2018.

[8] H. Kim, J. Lee, L. Eunkyu, and Y. Kim, "Touching the Untouchables: Dynamic Security Analysis of the LTE Control Plane," in *Proceedings of the IEEE Symposium on Security & Privacy (SP)*. IEEE, May 2019.

[9] S. R. Hussain, M. Echeverria, I. Karim, O. Chowdhury, and E. Bertino, "5GReasoner: A Property-Directed Security and Privacy Analysis Framework for 5G Cellular Network Protocol," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 669–684.

[10] W. Johansson, M. Svensson, U. E. Larson, M. Almgren, and 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*. IEEE, 2014, pp. 323–332.

[11] L. Xu, J. Wu, and C. Liu, "T3FAH: A TTCN-3 Based Fuzzer with Attack Heuristics," in *2009 WRI World Congress on Computer Science and Information Engineering*, vol. 7. IEEE, 2009, pp. 744–749.

[12] 3GPP. (2020) Evolved Universal Terrestrial Radio Access (E-UTRA); Radio Resource Control (RRC); Protocol specification. TS 36.331. Online. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2440.

[13] ——. (2020) NR; Radio Resource Control (RRC); Protocol specification. TS 38.331. Online. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3197.

[14] ITU. (2015, Aug) Recommendation ITU-T X.681 International Standard 8824-2: Information technology – Abstract Syntax Notation One (ASN.1): Information object specification. Online. https://www.itu.int/rec/T-REC-X.680-X.693-201508-I/en.

[15] S. R. Systems. (2020) srsLTE. Online. https://github.com/srsLTE/srsLTE.

[16] openLTE. (2020) openLTE. Online. https://sourceforge.net/projects/openlte.

[17] 3GPP. (2020) E-UTRA; Packet Data Convergence Protocol (PDCP) specification. TS 36.323. Online. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2439.

[18] A. Helin. (2020) Radamsa. Online. https://gitlab.com/akihe/radamsa.

[19] V. Velichkov. (2020) Fork of asn1c: Open Source ASN.1 Compiler. Online. https://github.com/velichkov/asn1c.

[20] OAI. (2020) ASN.1 extractor from 3GPP specifications. Online. https://gitlab.eurecom.fr/oai/openairinterface5g/-/blob/master/openair2/RRC/LTE/MESSAGES/asn1c/ASN1_files/extract_asn1_from_spec.pl.

[21] ——, "OpenAirInterface," 2020, Online. https://openairinterface.org/.