2014 IEEE International Conference on Software Testing, Verification, and Validation

T-Fuzz: Model-Based Fuzzing for Robustness Testing of Telecommunication Protocols

William Johansson, Martin Svensson, Ulf E. Larson
Ericsson AB
{*william.x.johansson,martin.x.svensson,ulf.t.larsson*}*@ericsson.com*

Magnus Almgren, Vincenzo Gulisano
Chalmers University of Technology
magnus.almgren@chalmers.se

***Abstract*—Telecommunication networks are crucial in today's society since critical socio-economical and governmental functions depend upon them. High availability requirements, such as the "five nines" uptime availability, permeate the development of telecommunication applications from their design to their deployment. In this context, robustness testing plays a fundamental role in software quality assurance. We present T-Fuzz – a novel fuzzing framework that integrates with existing conformance testing environment. Automated model extraction of telecommunication protocols is provided to enable better code testing coverage. The T-Fuzz prototype has been fully implemented and tested on the implementation of a common LTE protocol within existing testing facilities. We provide an evaluation of our framework from both a technical and a qualitative point of view based on feedback from key testers. T-Fuzz has shown to enhance the existing development already in place by finding previously unseen unexpected behavior in the system. Furthermore, according to the testers, T-Fuzz is easy to use and would likely result in time savings as well as more robust code.**

I. Introduction

Telecommunication networks and services are today classified as critical societal infrastructures [1], [2]. They fulfill socio-economical functions such as connecting end users over voice and text, and providing Internet access. According to [3], there are over 6.8 billion mobile cellular subscriptions, and over 2 billion active mobile-broadband subscriptions worldwide. Telecommunication also fulfills governmental functions, such as carrying emergency calls and public warning messages during a crisis.

To meet the needs of end users and governments alike (e.g., to make sure that emergency calls do not disappear between the caller and the dispatch), telecommunication operators must guarantee high availability of their networks. A common availability requirement is 99.999% uptime [4], which is less than five and a half minutes downtime per year. Considering these requirements, extended disturbances would soon put the operator out of business. To guarantee high availability, both software and hardware need to be robust against failures. Errors that are found in software running in production systems need to be fixed by applying patches, which in turn may require restarting software. This causes inevitable downtime and is not acceptable. Thus, a sound robustness testing strategy for testing the software is critical to ensure that errors are found early in the development process.

A well-known technique for robustness testing of software is fuzzing [5]. In fuzzing, further described in Section II-C, malformed messages are injected into a System Under Test (SUT). Malformed messages can be created from scratch (*generation-based* fuzzing) or as mutations of valid messages (*mutation-based* fuzzing). The SUT is monitored for unwanted behavior to determine if an injected message caused any problem. Fuzzing has earlier been used in traditional computing environments such as various UNIX platforms and network services [6]. However, recent works [7], [8], [9], [10], [11] show that fuzzing has been successfully used to find exploitable bugs also in the 2G and 3G cellular networks.

In this paper we present a methodology of how the robustness testing procedure for telecommunication products can be improved by also integrating fuzzing in the normal conformance testing. There are already existing robustness testing tools that specifically target the telecommunication area [12], [13], where certain protocol models and test cases are provided. However, to support new protocols, these either have to be provided by the vendor or written from scratch by the tester. Furthermore, these tools do not integrate into existing environments, making their use a completely separate testing task, often with the requirement of rather specialized knowledge to use the tool in question. Clearly, there are several advantages if the robustness testing could be integrated as part of the regular testing environment. It would then be executed as part of any test suite as well as being easier for the testing engineers to use.

We developed T-Fuzz – a novel fuzzing framework for telecommunication networks that overcomes the limitations of existing tools. T-Fuzz provides the following benefits.

- It integrates with existing testing environments.
- It easily adapts to different telecommunication protocols.
- It achieves full protocol coverage and
- it is easy to use.

These claims will be elaborated upon in the rest of the paper. Since T-Fuzz is integrated with the TTCN-3 (Testing and Test Control Notation Version 3) framework for confor-

978-0-7695-5185-2/14 $31.00 © 2014 IEEE
DOI 10.1109/ICST.2014.45

mance testing, further described in Section II-B, it achieves several advantages. The framework is general enough so that it can reuse models of 3GPP protocols that are created for conformance testing. As conformance testing is a mandatory activity of the SUT, such models are always created when new protocols are introduced. A side effect with the existence of such models is that fuzz testing can start very soon after a new protocol is introduced in the product, making the testing time efficient. Furthermore, having the complete model specification let us achieve full protocol coverage through *generation-based fuzzing*. Having the model integrated into a known environment and removing the tedious model creation task means testers should find it much easier to use than a separate tool.

T-Fuzz has been fully implemented and integrated into TITAN, a TTCN-3 compiler and execution environment. We describe the design and implementation of T-Fuzz. Through a modular design, minimum customization is needed when changing protocol or SUT. We describe the instantiation of T-Fuzz for a specific LTE protocol, the 3GPP Non-Access Stratum (NAS) protocol [14], and measure its usefulness in finding unexpected behaviors and discuss lessons learned. We also complement the technical discussion with a qualitative evaluation based on initial feedback from the conformance test team. Their feedback is presented together with observations on the usefulness of the framework from a test engineer perspective.

The remainder of this paper is outlined as follows. In Section II, we provide details of the telecommunication network, existing test environments in use, as well as a brief introduction to fuzzing. Given this background knowledge, we describe the design of T-Fuzz and its implementation in an existing test environment in Section III and Section III-A, respectively. In Section V we evaluate the usefulness of T-Fuzz. We describe the instantiation of T-Fuzz for the NAS protocol [14] with an evaluation of the unwanted behavior found. We then analyze the results from a quantitative questionnaire answered by key test engineers about the usefulness of T-Fuzz. Related work is presented in Section VI. We conclude the paper in Section VII.

II. Background

In this section we first provide the reader with background information about the LTE telecommunication network infrastructure, the Evolved Packet System. We continue discussing how conformance testing is performed in this context and present the TTCN-3 testing language. Finally, we briefly introduce the fuzzing testing technique.

A. The Evolved Packet System

The Evolved Packet System (EPS) provides LTE connectivity and is divided into three main networks: Evolved Universal Terrestrial Radio Access Network (E-UTRAN) for wireless communication, Evolved Packet Core (EPC) which handles LTE cellular communication [15] and Public Data Networks (PDN) for access to external networks such as the Internet. A brief overview of the EPS is illustrated in Figure 1.

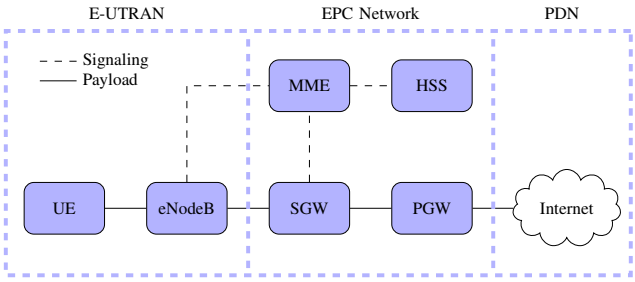

Figure 1. Simplified overview of the EPS network with the core nodes

E-UTRAN makes it possible for User Equipment (UE), such as mobile phones and tablets, to access the telecommunication network [16]. The UE communicates with a radio base station, referred to as E-UTRAN Node B (eNodeB), which provides signaling capabilities to the Mobility Management Entity (MME) and forwarding of payload messages from the UE towards the PDN. The MME is a core node in the EPS and is responsible for the attachment (connection procedure) and detachment (disconnect procedure) of UEs [17]. The MME assures the sustainability of UEs connection to PDNs by handling the handover between eNodeB stations. The Home Subscriber Server (HSS) provides the MME with authentication and authorization functions of an attaching UE. When the MME has authenticated the UE as a valid EPS network user, addresses to a Serving Gateway (SGW) and a Packet Data Network Gateway (PGW) are sent to the UE. The SGW and PGW are the two gateways an IP packet from E-UTRAN has to pass in order to reach a PDN.

B. Conformance Testing in EPS

The EPS network is built upon standardized interfaces and protocols to support an open market for telecommunication operators. Rigorous testing is required to ensure correct communication between two nodes from different vendors. Conformance testing is intended to test the product against the specified standards. In order to accomplish this, it requires the model representation of the protocol specification to be covered in full detail.

The TTCN-3 testing language is standardized by the European Telecommunications Standards Institute (ETSI) and commonly used for test specifications [18]. The TTCN-3 test system provides internal and external communication and requires test cases and models in TTCN-3 language [19], [20].

The TTCN-3 test language is strongly typed and contains a number of built-in types, such as `integer` for integers, `bitstring` for strings of bits, and `charstring` for

strings of characters. It also defines structured types, such as `record`, to specify structures of typed fields. A TTCN-3 module usually contains type definitions, port definitions, functions, and test cases. Type definitions are used to specify structures or to subtype any built-in type by adding restrictions (a subtype of `integer` may restrict the value to be in the range $0 - 255$, while a subtype of `bitstring` may restrict it to be exactly 5 bits long). Port definitions describe communication links and message's types. Lastly, test cases describe the test logic (e.g., SUT's ports to which to connect, messages to be sent or expected replies).

In order to perform tests of the system implementation, a test environment for the SUT has to be present with simulated surrounding nodes. TITAN is a TTCN-3 compiler and execution environment which generates C++ code from TTCN-3 [21]. The generated C++ code is compiled together with runtime libraries and external C++ source files that will be executed in the TTCN-3 test system. External C++ code is used to implement the communication link over the test ports as well as providing extra functionality to use from the TTCN-3 test cases. With this technique, a native standalone binary can be produced for good performance and executed on many different platforms.

C. Fuzzing

Fuzzing [22] is a software robustness testing technique. When fuzzed, the SUT is fed with malformed or invalid data and monitored for exceptions such as unexpected behavior. Fuzzing techniques are categorized as either mutation-based or generation-based [23]. A mutation-based fuzzer creates malformed or invalid data modifying valid messages with mutators. Several mutation techniques can be applied to modify valid messages: substitution, bit flipping, data removal or data addition, among others. One or more of these mutation techniques can be applied by a mutator to message fields or to specific bytes of valid messages. Opposite to mutation-based fuzzers, generation-based fuzzers build malformed messages from scratch and rely on message models with full protocol specifications. It should be noted that, depending on the SUT, malformed messages might require a valid checksum or ciphering. To this end, protocol awareness is required to provide good code coverage (e.g., to ensure malformed messages are not systematically discarded by the SUT due to wrong checksums).

III. Design

In this section, we present the design requirements and the main components of the T-Fuzz framework.

T-Fuzz is intended to extend the TITAN conformance test framework providing robustness testing by means of generation-based fuzzing. We aim at a full integration between TITAN and T-Fuzz since this would lead to a more extensive testing of new protocols used in products under development.

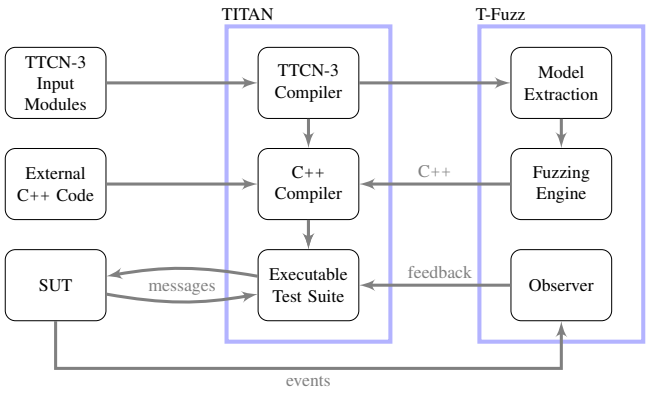

Figure 2. T-Fuzz components extending the conformance test environment

Our first design goal is to transparently integrate the functionality enabled by T-Fuzz with TITAN's ones. Doing this, it should be possible to fuzz all the communication protocols supported by TITAN. In order to achieve this, T-Fuzz should rely on the protocol models defined in TTCN-3, which are already used by TITAN to generate conformance test suites. Our second design goal is to rely minimally on a tester when defining a test suite where fuzzed messages are sent to the application under test. In order to achieve this, T-Fuzz should provide a way of creating fuzzed messages that resembles the one used to create valid ones. That is, the tester should only decide whether a message sent to the SUT is valid or fuzzed, but he should not be required to specify how to fuzz the message itself.

Since we rely on the generation-based fuzzing approach, we can create fuzzed messages by generating arbitrary data in any field of any protocol message. As discussed in [23], this greatly enhances the strength of robustness testing compared to mutation-based fuzzers such as T3FAH [24]. Furthermore, it would also result in a higher code coverage since it could generate all the different messages specified by a given protocol.

A. T-Fuzz: An Extension of the TITAN Environment

Figure 2 presents the TITAN environment extended with the T-Fuzz framework. In the following, we provide a description of the TITAN environment and continue discussing T-Fuzz's composing modules. When designing a new test suite, a set of TTCN-3 protocol models and user specified test cases is fed to the TITAN TTCN-3 Compiler. The latter parses and analyzes the input modules in order to generate C++ code to communicate with the SUT. The external C++ code, provided by the tester, implements the interface towards the SUT and, once compiled together with the generated code, results in the final Executable Test Suite (ETS). The generated ETS contains the necessary functionality for setting up and executing the test (e.g., encoding and decoding of messages or logging). During

```
module Example
{
    type bitstring BIT4 length(4);
    type hexstring HEX2 length(2);
    type record MSG
    {
        BIT4    fourBitField,
        HEX2    twoHex,
        integer number optional
    };
}
```

Figure 3. Example TTCN-3 module with type definitions

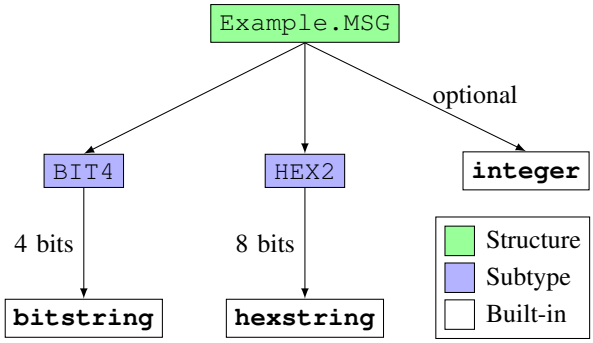

Figure 4. Extracted model tree of MSG structure in Example module

execution, test messages are sent to the SUT, which responds accordingly. At the end of an execution run, a summary with verdicts of the test cases is presented to the tester.

As discussed in the previous section, we aim at a transparent integration between TITAN and T-Fuzz. Given the steps required to create an ETS, we would like to provide auto-generated functions that can be used to inject fuzzed messages to the SUT. The framework should also report results once a robustness test has been completed.

As presented in Figure 2, T-Fuzz is composed of three main modules:

a) **Model Extractor**: used to extract TTCN-3 models that are later used as input for the fuzzer.
b) **Fuzzing Engine**: used to produce C++ code that provides functions to generate fuzzed messages according to the extracted models.
c) **Observer**: used to monitor the SUT and provide feedback to the ETS to take online decision about how to fuzz the SUT.

We present each module in detail in the following paragraphs.

Model Extractor: Since T-Fuzz relies on generation-based fuzzing, it must be able to generate messages from scratch. Nevertheless, in its current implementation, there is no built-in functionality in TTCN-3 to randomly generate messages. In order to do this, the models defined in TTCN-3 are extracted by the Model Extractor. The latter is able to extract all subtypes and structured types from the input models statically at compile time. It is also capable of distinguishing between optional or mandatory structure fields. Finally, it only requires input model's definitions and does not need populated instances of them.

Fuzzing Engine: Once the models for the messages defined by a protocol have been extracted, the Fuzzing Engine provides the functionality to generate instances of the models. This is achieved by taking the extracted models and creating generation functions for all the built-in TTCN-3 types (see Section II-B), the extracted subtypes and structured types. It also provides functions to specify which structure fields should not be randomized but rather set to a specific value (e.g. identification fields). Finally, it allows to specify a fixed seed as basis for the randomization in order to have repeatable experiments.

The Observer: The purpose of the Observer is to monitor the SUT and provide feedback to the ETS to guide the fuzzing depending on the observed results. Different to the Model Extractor and the Fuzzing Engine, the specific Observer implementation depends on the SUT. The current plan for T-Fuzz is to provide Observer templates for commonly monitored system features such as CPU usage, memory consumption of the SUT or logs. We discuss a specific implementation of an Observer in the evaluation (Section V-B).

IV. Implementation

In this section, we discuss in more detail and present the implementation of the *Model Extractor* and *Fuzzing Engine* modules. As discussed in Section III-A, the implementation of the *Observer* module depends on the SUT, we present a possible implementation in Section V-B. We conclude the section discussing the TTCN-3 API provided by T-Fuzz to tune how the SUT is fuzzed.

A. Model Extractor

With TITAN, the input TTCN-3 model files are parsed to generate C++ code that is later used to connect and forward messages to the SUT. The *Model Extractor* has been developed as an extension of the TITAN's TTCN-3 Compiler. After the C++ code has been generated, the *Model Extractor* statically analyzes it in order to extract the information needed to create fuzzed messages.

While parsing the code generated by TITAN, the *Model Extractor* builds a model tree of structures where each field is represented by a node. These nodes describe the types of the fields, where subtypes are represented by internal nodes while leaf nodes represent one of the seven built-in TTCN-3 types: `boolean`, `integer`, `float`, `charstring`, `bitstring`, `hexstring` or `octetstring`. In many protocols, especially in binary

ones, subtypes define a length requirement on built-in types. As an example, `type bitstring BIT4 length(4)` would define a subtype of type `bitstring` whose length is of exactly four bits. Furthermore, fields in a `record` structure can be specified as mandatory or optional. The *Model Extractor* parses all existing fields taking into account existing restrictions and distinguishing between mandatory and optional ones. A sample module containing subtype and structure definitions is presented in Figure 3. The module `Example` defines two subtypes (`BIT4` and `HEX2`) and one structure (`MSG`) composed by the same subtypes and by an optional built-in `integer`. The resulting extracted model for the TTCN-3 structure `MSG` is shown in Figure 4. Once model trees have been created, they're fed to the *Fuzzing Engine* module, as discussed in the following section.

B. The Fuzzing Engine

The generation of data takes place in the *Fuzzing Engine*. The latter relies on the models extracted by the *Model Extractor* in order to create instances of the different structures and subtypes. The resulting generation code is provided to TITAN and compiled to the final ETS.

Generation of structure fields is made by traversing the model tree and generating values for each of the descendant nodes. While parsing the nodes, fields marked as optional are instantiated randomly. As discussed in Section II-C, generation-based fuzzing might result in poor code coverage if specific message parts are not created accordingly to the fuzzed values of a message. For this reason, the *Fuzzing Engine* allows the tester to specify which fields should be set to specific values rather than fuzzed ones. All the different fields are first looked up in an override table before their value is fuzzed. If a value exists in the table, this value is used instead of a fuzzed one. The TTCN-3 environment will be used to perform operations on the messages and encoding them (i.e., calculate checksums and perform ciphering). The `union` and `enum` types will be generated by choosing one possible alternative at random. The built-in types are generated by assigning bits randomly. The random bits returned are based on a random seed that can be set at any time. This way, messages resulting in unexpected behavior can be easily repeated.

C. TTCN-3 API

T-Fuzz relies on the TTCN-3 language. By returning the data generated in T-Fuzz expressed in TTCN-3, the existing infrastructure used for conformance testing can be transparently used to fuzz the SUT. After model extraction and creation of the generators in the *Fuzzing Engine*, T-Fuzz creates a TTCN-3 module that serves as the glue between the TTCN-3 input models and the C++ code. This module exposes all the generator functions created by the fuzzing engine for all subtypes, structures and built-in types using

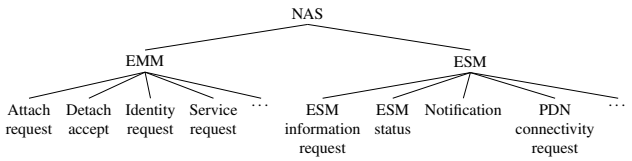

Figure 5. Tree over NAS protocol with a subset of EMM's and ESM's message types

the format `fuzz_gen_<M>_<T>()`, where `M` and `T` represent the *model* and the *type*, respectively. These functions can be used by the tester to control how messages are fuzzed. With respect to the *Example* module in Figure 3, generators for all defined subtypes (`BIT4` and `HEX2`) and structures (`MSG`) are exposed. A randomly generated `BIT4` can therefore be created by calling `fuzz_gen_Example_BIT4()` while a random `MSG` structure can be created invoking `fuzz_gen_Example_MSG()`.

As described in the previous section, some fields need to be preserved from random generation. Functions to save these values are also exposed in the glue code in the format `fuzz_override_<M>_<T>(name,val)`. With respect to the *Example* module (Figure 3), a call to `fuzz_override_Example_HEX2("twoHex", 'FA'H)` will ensure that all generated structures containing a `HEX2` field named `twoHex` will set its value to 0xFA.

V. Evaluation

In this section, we evaluate the T-Fuzz framework from both a technical and a qualitative perspective (based on feedback from key testers of the SUT).

In the first part of the evaluation, we present how T-Fuzz has been used to perform robustness testing on the NAS protocol [14], used in LTE telecommunication networks to carry signaling messages between a UE and the SUT[1]. We focus on NAS since it is used in conformance testing of the SUT and the necessary protocol models are already available. We first present an overview of the protocol and then proceed describing the evaluation setup and the testing outcome.

The second part of the evaluation discusses the results of a survey that has been conducted as part of the evaluation together with key testers of the SUT. The survey focuses on the usability and possible benefits enabled by T-Fuzz.

A. NAS Overview

The NAS specification defines two distinct protocols for EPS Mobility Management (EMM) and EPS Session Management (ESM) between a UE and the SUT. The SUT handles mobility functionality (e.g., attach and detach of a UE, authentication and ciphering setup) and specifies 32

[1]The actual system under test is not disclosed. As our focus is on the testing methodology, there is no loss of generality.

different message types [14]. ESM handles establishment and control of user data (e.g., connection to a PDN or establishment of bearers) and specifies 22 different message types. An example of the NAS message tree is shown in Figure 5.

EMM and ESM protocols are defined and implemented by means of state machines. Both the UE and the SUT have one EMM state machine and one ESM state machine per bearer context. We focus further on the description of the EMM state machine, which is responsible for the mobility of the UE, since ESM state machines are only responsible of the PDN connectivity and bearers which do not affect the communication with the SUT. A complete description of the EMM's and ESM's state machines can be found in [14].

The SUT's EMM state machine is based around the mobility of the UE. As presented in Figure 6, the main states are EMM-REGISTERED and EMM-DEREGISTERED and specify whether the UE is attached or detached to the network, respectively. In order for a UE to be attached, an *Attach Request* message is sent to switch from state EMM-DEREGISTERED to EMM-REGISTERED-INITIATED and is followed by an *Attach Accept* message in order to switch to the EMM-REGISTERED state.

A UE can be in either EMM-IDLE or EMM-CONNECTED mode, depending on whether a connection to the MME is properly setup. When in EMM-IDLE mode, the UE is allowed to send five different messages (referred to as *initial* NAS messages) to switch to the EMM-CONNECTED mode. When in EMM-CONNECTED mode (i.e., when a proper signaling connection exists between the UE and the SUT), several more messages are allowed to be sent and received, depending on the EMM state. According to the standard [14], most actions are performed in the EMM-REGISTERED state. That is, most of the messages are sent after a complete attach sequence (EMM-REGISTERED state) and an initial message that switches the UE from EMM-IDLE to EMM-CONNECTED mode.

B. Evaluation Setup

The SUT testers has a complete environment in place for conformance testing of the NAS protocol. By using this existing environment, we could readily setup the necessary network equipment (i.e., the UEs and the SUT). As discussed in Section V-A, the messages that can be sent by the UE depend on its state. To this end, the SUT testing environment eases the fuzzing of the NAS protocol by allowing us to start an experiment by sending the valid messages required to reach the desired UE's initial state. The SUT is completely isolated and the fuzzing tests will be the only executions affecting the SUT.

The SUT is implemented largely in the Erlang programming language. An Erlang application is built by using strongly isolated lightweight processes which share no memory, with message passing as the only interaction between

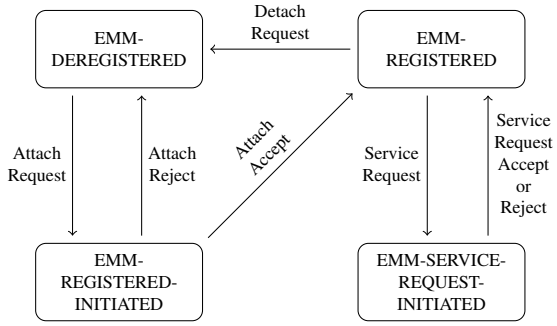

Figure 6. Simplified EMM state machine

them [25]. According to the Erlang philosophy, functionality should be divided into supervisor trees, where each process node is either a supervisor or a worker [26]. The supervisor processes manage their child processes, either workers or supervisors, and monitor if they are alive. The worker processes only execute a single task. In the SUT, every attached UE is represented by an Erlang worker process. If an unwanted state is entered during fuzzing, the UE is disconnected. Such a condition does not necessarily imply a bug in the system since it might have been deliberately provoked as the only way to handle an event, but it can be an indicator that further investigation is needed.

As specified in Section III-A, two out of the three modules of T-Fuzz (the *Model Extractor* and the *Fuzzing Engine*) have a general implementation and do not need to be tailored to each specific protocol or SUT. Only the templates of the T-Fuzz Observer need to be changed to the environment of the SUT. For this evaluation, we have implemented the T-Fuzz Observer as an Erlang counter that keeps track of the number of unwanted states that are entered during an experiment. The Observer will keep record of all messages and resend the ones potentially causing unwanted behavior. This is to guarantee reproducibility. At the end of each experiment, the Observer generates a report that associates each unwanted behavior with the fuzzed input message that caused it. The resulting reports have been shared with the testing team for further investigation.

C. NAS Protocol Fuzzing

Four different test cases are defined for our evaluation: $\mathbf{Initial}$, $\mathbf{Phase_1}$, $\mathbf{Phase_2}$ and $\mathbf{Phase_3}$. The first test case, $\mathbf{Initial}$, is intended to send malformed messages from the UE to the SUT before the former is attached to the latter. The other test cases perform different kinds of setup, with different states being entered. These test cases extend the previous one by adding more complex setup and configuration, thus letting us focus on the fuzzing activities to a particular setup to see if any unwanted behavior is shown then.

T-Fuzz was tested over a couple of weeks for these 4

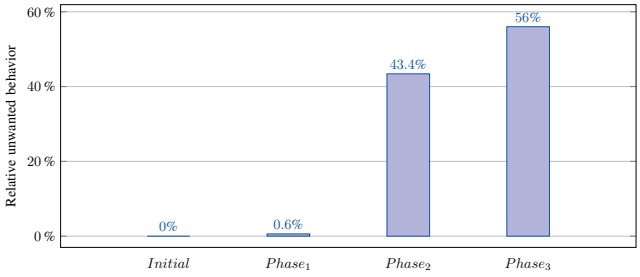

Figure 7. Unwanted behavior observed in the NAS test cases, relative to the total amount of unwanted behavior

Figure 8. The left figure shows the percentage of the message types that are involved in causing unwanted behavior, while the right figure highlights the origin of these for the $\mathbf{Phase_3}$ test case

different test cases, sending the same number of messages for each test case. The bar plot in Figure 7 shows the amount of unwanted behavior that is observed for each test case, with respect to the total number of unwanted behaviors. The $\mathbf{Phase_2}$ and $\mathbf{Phase_3}$ test cases together counts for the majority of the unwanted behaviors.

Further statistics are collected and presented for the $\mathbf{Phase_3}$ test case, which was the origin of the most unwanted behaviors. We analyzed which NAS messages were involved. The portion of message types involved, presented in the left chart in Figure 8, shows that only 26.8% of the types were represented. This may well be used for further analysis on what parts of the implementation to focus testing on. The portion of messages that should, according to the standard [14], originate from the UE or the SUT respectively is presented in the right chart in Figure 8. The result show that the messages originating from the SUT are slightly overrepresented.

The results may indicate that the messages causing unwanted behavior were unexpected and could not be properly handled, or that the worker process entered a state that called for the supervisor to interrupt its execution because the UE sent a message it should not. Results also indicate that the UE and the SUT have different views on the connection states and the easiest way to solve a conflict might be to simply restart the communication. The resulting statistics can be used to decide what parts of the SUT should be further investigated. Since the messages are completely reproducible, it is easy to troubleshoot an unwanted behavior by simply sending the same messages again.

Table I
ANSWERS FROM QUESTIONNAIRE

Question	Answer
$\mathbf{Q_1}$	Average 2.3 out of 5
$\mathbf{Q_2}$	Average 4.6 out of 5
$\mathbf{Q_3}$	Average 4.2 out of 5
$\mathbf{Q_4}$	Average 2 out of 5
$\mathbf{Q_5}$	Average 3.6 out of 5
$\mathbf{Q_6}$	Median 1.75 days
$\mathbf{Q_7}$	Average 4.17 out of 5
$\mathbf{Q_8}$	1/3 yes, 2/3 no
$\mathbf{Q_9}$	Average 4 out of 5
$\mathbf{Q_{10}}$	Median 10%
$\mathbf{Q_{11}}$	Median 4.5 days

D. Observations on Generalizability and Usability

T-Fuzz is at this stage very capable of testing the 3GPP protocols. Several protocols are modeled in TTCN-3, and testing them with T-Fuzz is very straight-forward. The time needed to implement fuzzing of a new protocol depends on the amount of required functionality (e.g., to reach a wanted state before fuzzing, to evaluate response messages and so on). Plainly sending fuzzed messages to the SUT takes no time to setup. This has been proven to work with initial testing of another 3GPP protocol without much test logic, but could easily be made more efficient by an experienced tester with knowledge about the protocol and its states.

When the design and implementation of T-Fuzz was ready, we performed an evaluation of its usability and applicability within the organization. The purpose of this evaluation is to report the internal experience with the tool, and also to gather information that other testers in the same field may benefit from. For the evaluation, we gathered the testers that work with TTCN-3 and TITAN towards the SUT, i.e. the key testers. We demonstrated T-Fuzz for them and we also presented and discussed unwanted behavior observed while fuzzing the NAS protocol. We then asked the testers to fill in a questionnaire regarding T-Fuzz. The following eleven questions were given.

- $\mathbf{Q_1}$: Would this tool save time in your daily activities? [1-5]
- $\mathbf{Q_2}$: Would this tool be helpful to discover hidden/new aspects of the product? [1-5]
- $\mathbf{Q_3}$: Would this tool increase the quality of the product? [1-5]
- $\mathbf{Q_4}$: Would this tool require a fuzzing expert to use? [1-5]
- $\mathbf{Q_5}$: Would this tool fit with the testing requirements? [1-5]
- $\mathbf{Q_6}$: How many days would it take to get familiar with the tool? [Number of days]
- $\mathbf{Q_7}$: How likely is it that you would use such a tool, given that you should perform negative testing? [1-5]
- $\mathbf{Q_8}$: Have you found unwanted behavior similar to the one presented? [Yes/No]
- $\mathbf{Q_9}$: Would this tool save time in finding such unwanted behavior? [1-5]
- $\mathbf{Q_{10}}$: What is the probability the unwanted behavior would

be found by you without this tool? [Percentage]

$\mathbf{Q_{11}}$: How many days would it take to find this unwanted behavior without the presented tool? [Number of days]

The answers (presented in Table I) provided us with interesting input about the usefulness and ease of use of T-Fuzz. Question $\mathbf{Q_1}$ got a rather negative response, 2.3 on average. The reason is that questioned testers do not perform this kind of negative testing in their daily activities. Nevertheless, an average of 4.6 for question $\mathbf{Q_2}$ shows that testers believe T-Fuzz is capable of finding bugs which would not have been found with current test methods. Question $\mathbf{Q_3}$ confirms, with an average of 4.2, that the quality of the product would improve using T-Fuzz as a complement of the existing robustness testing already performed. The answers to questions $\mathbf{Q_4}$ (2 on average) and $\mathbf{Q_6}$ (a median of 1.75 days) show that the testers do not feel a deep knowledge of fuzzing is required to use T-Fuzz. Testers believe T-Fuzz is easy to use and they could get familiar with it after 1-3 days. The average 3.6 on question $\mathbf{Q_5}$ indicates that the tool would fit with the existing testing requirements. Distinguishing the testing between positive and negative, the testers' average 4.17 for question $\mathbf{Q_7}$ indicates that they believe the tool would be used.

Questions $\mathbf{Q_8}$ to $\mathbf{Q_{11}}$ relate to the unwanted behavior presented to the testers. The answers for question $\mathbf{Q_8}$ indicate that similar unwanted behavior are rarely found. One third of the testers have unintentionally encountered similar unwanted behavior while testing other parts of the SUT. This is further supported by the median value 10% for question $\mathbf{Q_{10}}$, which indicates that the presented unwanted behavior would probably not have been found early in the testing phase. The average 4 for question $\mathbf{Q_9}$ states that the key testers of the SUT believe T-Fuzz would simplify the task of finding such unwanted behavior. Question $\mathbf{Q_{11}}$ gives an estimation of the time saved using T-Fuzz instead of finding the unwanted behavior manually. By using T-Fuzz, the median value states that over 4.5 days could be saved.

The outcome of the survey highlights that T-Fuzz can contribute in solving some of the challenges presented in [27], as described in Section VI. The key challenge of being able to perform robustness testing of smaller parts of the system is addressed by T-Fuzz with specific per-protocol tests. This also simplifies troubleshooting knowing in which part of the overall system a fault has occurred. By using T-Fuzz, one test case can create a large amount of messages leading to good coverage of the protocol. This addresses the challenge that relies in the creation and selection of which robustness tests should be performed. As for the challenge of creating a representative testing environment, making use of TITAN's conformance test environment ensures that T-Fuzz resembles as much as possible the customer's environment. T-Fuzz also provides the possibility of efficiently performing robustness testing continuously and automatically, which addresses the challenge of manually executing test cases with large time constraints.

VI. Related Work

In this section, we discuss the existing work related to fuzzing and robustness testing in the context of telecommunication industries. We also present existing fuzzing environments and discuss why the fuzzing component of T-Fuzz is implemented as a stand-alone module.

Fuzzing and robustness testing in telecommunication industries: The authors of [27] present a case study of the existing robustness testing strategies and practices defined in a telecommunication industry (Ericsson AB). As discussed, the main bottleneck in robustness testing is not the lack of test cases but rather the long times and high costs required to properly setup a testing environment. A complete testing environment involves real hardware, the prototype or under-development application that will be tested and simulated or real input data. Several features need to be monitored during robustness testing, ranging from hardware to software issues. Moreover, it is also generally difficult to systematically create and select which test cases to perform in order to ensure good testing coverage. To this end, T-Fuzz aims at providing a transparent and time-saving framework for robustness testing (by means of generation-based fuzzing) relying on the hardware and software tools that are already used for conformance testing.

In [24], the authors present T3FAH: a mutation-based fuzzer that, similarly to T-Fuzz, relies on TTCN-3 to extract the SUT's input model. With T3FAH, real data being sent to the SUT is mutated in an on-line fashion. Mutation rules are defined by means of an *attack pattern library* and are generated modifying input fields with suitable mutation operators. The latter are type dependent generators that can also fuzz fields trying to exploit common vulnerabilities. As an example, a buffer overflow attack pattern on the TTCN-3 `charstring` type could mutate such a field with a 1024 bytes long random string. T-Fuzz differs from T3FAH in several aspects. First, T3FAH case study focuses on the fuzzing of SIP clients while we focus on the NAS protocol. Secondly, being mutation-based, it requires the tester to specify how input messages should be mutated (i.e., to specify the attack pattern library). Finally, the presented work does not discuss how T3FAH can be integrated in existing testing environments, which is one of the main contributions of T-Fuzz.

The DIAMONDS project [28] focuses on model-based security testing, a research field that studies how to systematically and efficiently define security test objectives. Some work in the DIAMONDS project has been focused on the generation of fuzzing tests that rely on the TTCN-3 testing language. The authors state that generators and mutators should be implemented by making use of already existing fuzzing frameworks, such as Peach [29], in order to ease the development of fuzzing tests. As discussed in [30], it

is possible to test new protocols with reasonable effort by relying on existing fuzzing frameworks. While T-Fuzz shares some commonalities with the DIAMONDS project, it can automatically generate values for all models in TTCN-3 without relying on third party generators or mutators. We discuss in the following paragraph why T-Fuzz does not rely on existing fuzzing frameworks.

Major fuzzing frameworks: Some of the existing fuzzing frameworks that are most commonly used include the Peach Fuzzer [29], Radamsa [31], Codenomicon Defensics [12] and the P1 Telecom Fuzzer [13]. The Peach Fuzzer [29] is probably the most common one and provides both mutation-based and generation-based fuzzing. Based on user-defined input models (Peach Pit files), it can generate messages both randomly or trying to exploit known vulnerabilities (as for T3FAH [24]). Differently from Peach, Radamsa [31] focuses only on mutation-based fuzzing. It requires a small effort to be set up since it operates on given sample inputs but does not result in large test coverage. Codenomicon Defensics [12] is a commercial model-based fuzzing tool which contains over 200 protocols of which 13 are telecommunication specific, such as GTPv1 and Diameter [32]. However, there is currently no support for LTE application layer protocols, such as S1AP or NAS. Another model-based fuzzing tool that specifically targets telecommunication networks is the P1 Telecom Fuzzer [13]. It supports a wide range of protocols for GSM, CDMA, WCDMA and LTE with support for some application layer protocols such as S1AP and NAS.

The main drawback of all these existing fuzzing frameworks is that protocol models have to be provided by the vendor or written from scratch by the tester. The amount of functional logic needed to execute the tests is huge, which makes creating a whole new test platform a very time consuming operation. Rather than relying on third party frameworks, T-Fuzz makes use of models already defined for conformance testing, thus not requiring the intermediate modeling step. In contrast to the third party frameworks, T-Fuzz is not a standalone tool that requires its own environment but instead uses the existing environments for executing conformance tests.

VII. Conclusions and Future work

In this paper we presented T-Fuzz, a novel fuzzing framework for robustness testing that integrates with a conformance testing environment. T-Fuzz extends the TITAN environment and allows for a fast development of robustness tests by relying on existing protocol models used in conformance tests. That is, it allows testers to fuzz new applications as soon as protocols (which are currently specified for conformance testing) are in place.

We have evaluated T-Fuzz with respect to the NAS protocol implementation and showed that it is able to provoke unwanted behavior of the SUT and to provide insights about which parts of SUT should be further investigated by the developers. We also conducted a survey with key testers focused on T-Fuzz generalizability and usability. The outcome of the questionnaire shows that T-Fuzz is perceived as easy to use (on average testers affirm 2.5 days are required to get familiar with it). The questionnaire also highlights that the majority of the testers would use it during normal work and that it would be useful to spot problems that would not be found exclusively relying on the current testing environment. Overall, it would be a valuable addition to the robustness testing activities.

As discussed, T-Fuzz allows for robustness tests to begin with a series of valid messages in order to reach the desired initial protocol state as this has a significant impact on the fuzzing outcome. In its current implementation, these initial steps are specified manually. We plan to improve this by automating the process for future test cases. The current version is a first step to see if the methodology works. We plan to increase T-Fuzz fuzzing logic to better fit textual protocols that, differently from binary ones such as NAS, define fields with variable length and require more logic behind the message generation. More advanced generation techniques, such as presented in [33], could further be implemented in T-Fuzz.

Acknowledgment

We would like to thank Ericsson AB for the opportunity of performing this project. This work has been partially supported by the European Comission Seventh Framework Programme (FP7/2007-2013) through the SysSec Project, under grant agreement 257007, through the FP7-SEC-285477-CRISALIS project and through the collaboration framework of Chalmers Energy Area of Advance.

References

[1] J. Moteff, C. Copeland, and J. Fischer, "Critical infrastructures: what makes an infrastructure critical?" DTIC Document, 2003.

[2] Commisson of the European Communities, "Critical Infrastructure Protection in the fight against terrorism." [Online]. Available: http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=COM:2004:0702:FIN:EN:PDF

[3] International Telecommunication Union, "Key ICT indicators for developed and developing countries and the world." [Online]. Available: http://www.itu.int/en/ITU-D/Statistics/Documents/statistics/2013/ITU_Key_2005-2013_ICT_data.xls

[4] J. Gray and D. P. Siewiorek, "High-availability computer systems," *Computer*, vol. 24, no. 9, pp. 39–48, 1991.

[5] A. Takanen, J. D. Demott, and C. Miller, *Fuzzing for software security testing and quality assurance*. Artech House, 2008.

[6] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services*. University of Wisconsin-Madison, Computer Sciences Department, 1995.

[7] F. Ricciato, A. Coluccia, and A. D'Alconzo, "A review of DoS attack models for 3G cellular networks from a system-design perspective," *Computer Communications*, vol. 33, no. 5, pp. 551–558, 2010.

[8] C. Mulliner, N. Golde, and J.-P. Seifert, "SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale," *USENIX Security Symposium*, 2011.

[9] R.-P. Weinmann, "All your baseband are belong to us," 2010. [Online]. Available: http://2010.hack.lu/archive/2010/Weinmann-All-Your-Baseband-Are-Belong-To-Us-slides.pdf

[10] Grugq, "Base Jumping: Attacking the GSM baseband and base station," 2010. [Online]. Available: http://www.coseinc.com/en/index.php?rt=download&act=publication&file=Base%20Jumping.pdf

[11] S. M. Harald Welte, "OsmocomBB: Running your own GSM stack on a phone," 2010. [Online]. Available: http://events.ccc.de/congress/2010/Fahrplan/attachments/1771_osmocombb-27c3.pdf

[12] Codenomicon Ltd., "Codenomicon Defensics," 2013. [Online]. Available: http://www.codenomicon.com/defensics/

[13] Priority One Security, "P1 Telecom Fuzzer (PTF)," 2013. [Online]. Available: http://www.p1sec.com/corp/products/p1-telecom-fuzzer-ptf/

[14] 3GPP, "3GPP TS 24.301 V12.0.0: Non-Access-Stratum (NAS) protocol for Evolved Packet System (EPS)." [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/24301.htm

[15] E. Dahlman, S. Parkvall, and J. Sköld, *4G: LTE/LTE-Advanced for Mobile Broadband*. Academic Press, 2011.

[16] 3GPP, "3GPP TS 23.401 V11.6.0: General Packet Radio Service (GPRS) enhancements for Evolved Universal Terrestrial Radio Access Network (E-UTRAN) access." [Online]. Available: http://www.3gpp.org/ftp/specs/html-INFO/23401.htm

[17] 3GPP, "3GPP TS 23.002 V12.2.0: Network architecture." [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/23002.htm

[18] ETSI, "Introduction – About TTCN-3," 2013. [Online]. Available: http://www.ttcn-3.org/index.php/about/introduction

[19] ETSI, "201 873-5 Part 5: TTCN-3 Runtime Interface (TRI), Version: 4.4.1," 2012. [Online]. Available: http://www.etsi.org/deliver/etsi_es/201800_201899/20187305/04.04.01_60/es_20187305v040401p.pdf

[20] ETSI, "201 873-6 Part 6: TTCN-3 Control Interface (TCI), Version: 4.4.1," 2012. [Online]. Available: http://www.etsi.org/deliver/etsi_es/201800_201899/20187306/04.04.01_60/es_20187306v040401p.pdf

[21] J. Z. Szabó and T. Csöndes, "TITAN, TTCN-3 test execution environment," *Infocommunications Journal*, vol. 62, no. 1, pp. 27–31, 2007.

[22] B. Miller, "Fuzz Testing of Application Reliability," 1988. [Online]. Available: http://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf

[23] C. Miller and Z. N. J. Peterson, "Analysis of Mutation and Generation-Based Fuzzing," 2007. [Online]. Available: http://securityevaluators.com/files/papers/analysisfuzzing.pdf

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

[25] J. Armstrong, "Concurrency Oriented Programming in Erlang," *Invited talk, FFG*, 2003.

[26] Ericsson AB, "Erlang, OTP Design Principles User's Guide, Overview," 2013. [Online]. Available: http://www.erlang.org/doc/design_principles/des_princ.html

[27] S. Eldh and D. Sundmark, "Robustness testing of mobile telecommunication systems: A case study on industrial practice and challenges," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 895–900.

[28] I. Schieferdecker, J. Großmann, and M. Schneider, "Model-Based Security Testing," in *MBT*, ser. EPTCS, A. K. Petrenko and H. Schlingloff, Eds., vol. 80, 2012, pp. 1–12.

[29] Michael Eddington, Deja vu Security, "Peach Fuzzer," 2013. [Online]. Available: http://peachfuzzer.com/

[30] DIAMONDS Consortium, "Development and Industrial Application of Multi-Domain Security Testing Technologies," 2013. [Online]. Available: http://www.itea2-diamonds.org/_docs/caseStudies/Case_Study_Experience_Sheet_Ericsson.pdf

[31] OUSPG, "Radamsa." [Online]. Available: https://code.google.com/p/ouspg/wiki/Radamsa

[32] Codenomicon Ltd., "Codenomicon Defensics for 3G/4G LTE," 2013. [Online]. Available: http://www.codenomicon.com/defensics/3g-4g-lte/

[33] T. Y. Chen, H. Leung, and I. Mak, "Adaptive random testing," in *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making*. Springer, 2005, pp. 320–329.