CHALMERS

Extending TTCN-3 with Model-Based Fuzzing for Robustness Testing of Telecom Protocols

Master of Science Thesis in Computer Systems and Networks

William Johansson
Martin Svensson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, October 2013

Extending TTCN-3 with Model-Based Fuzzing for Robustness Testing of Telecom Protocols

William Johansson
Martin Svensson

Examiner: Magnus Almgren
Supervisor: Vincenzo Gulisano and Ulf E. Larson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden October 2013

Abstract

The telecommunication network is classified by governments as a critical infrastructure which must be protected. It provides text and voice communication, Internet access, and emergency services for mobile subscribers worldwide. Operators set high demands on the availability of the telecommunication products and a common level to mark high availability is 99.999%, or less than five and a half minutes of downtime a year. Hence, telecommunication vendors have to thoroughly test their products to ensure that the demands are met. One way to achieve this is to apply a robustness testing technique called fuzzing.

In this master thesis we designed and implemented a model-based fuzzer for robustness testing of telecommunication protocol implementations. Our fuzzer is generation-based and integrates with the TTCN-3 conformance test environment by extracting protocol models and creates generators to populate the models. A case-study is conducted of fuzzing a telecommunication protocol which shows that the fuzzer is capable of provoking erroneous behavior, some which unlikely would have been found otherwise. After discussion with the conformance test team, the tool is considered easy to learn, and that it will be a helpful addition to the tester's toolbox. Taken together, we believe that the fuzzer will be a valuable asset for robustness testing.

Sammanfattning

Regeringar världen över klassificerar telekommunikationsnätverket som en kritisk infrastruktur som måste skyddas från attacker. Nätverket möjliggör kommunikation genom textmeddelanden, mobilsamtal, internetanslutning och nödsamtal för mobilabonnemang världen över. Höga krav sätts på operatörer att deras system ska ha en hög tillgänglighet där en välkänd nivå för en hög tillgänglighet är 99,999% vilket motsvarar mindre än fem och en halv minut per år då systemen inte är i drift. Därför måste tillverkare av telekommunikationssystem genomföra utförliga tester av systemen för att nå dessa höga krav. En metod som kan appliceras för att nå dessa mål är en teknik för robusthetstestning vid namn fuzztestning.

I detta examensarbete har vi designat och implementerat en modellbaserad fuzzer för robusthetstestning av protokollimplementationer inom telekommunikation. Vår fuzzer är baserad på meddelandegenerering och är integrerad i en konformitetstestmiljö, skriven i TTCN-3, genom att extrahera protokollmodeller och skapa funktioner av dessa som genererar värden för samtliga fält av de extraherade protokollen. Det utfördes också en undersökning där fuzzern testades gentemot telekommunikationsprotokoll där resultatet visade att fuzzern är kapabel till att hitta felaktiga beteenden, beteenden som troligtvis inte hade blivit funna på annat sätt. Efter diskussioner med konformitetstestgruppen ansågs det att verktyget är lätt att lära sig och att det skulle vara en värdefull tillgång för robusthetstestning.

Acknowledgement

This Master's thesis has been suggested and funded by Ericsson AB. The authors would like to thank everyone involved in the thesis. Big thanks to Ericsson and especially the SGSN-MME teams for helping us with everything.

Special thanks go out to our supervisor at Ericsson, Ulf E. Larson, for supporting and guiding our work from day one. Big thanks to our examiner, Magnus Almgren, and supervisor, Vincenzo Gulisano, at Chalmers which have been very helpful with providing feedback and presenting new ideas to our work. The authors would also like to thank everyone that has taken their time to proofread our thesis and come up with valuable comments.

Contents

Chapter 1

Introduction

The telecom network is a very important infrastructure in today's society. It provides voice and text communication, Internet access, and emergency services for mobile subscribers. According to the International Telecommunication Union (ITU), there are over 6.8 billion mobile subscribers and close to 2.1 billion mobile-broadband subscribers worldwide [1]. These subscribers require a connection to the telecom network to be able to utilize their subscriptions. By introducing Internet access via the telecom network, new technologies, such as the smartphone, arose. Smartphones need a connection to the telecom network for Internet access to be fully utilized. With an increasing usage of the telecom network the demands of its availability will also be higher. Today's society is already heavily dependent upon telecommunication services, where mobile communication over distance is assumed to be working. It is also a crucial way for communication and coordination in times of crisis. The US government and the EU commission both classify telecommunication infrastructure as a critical infrastructure [2, 3] which must be protected. The European commission describes a critical infrastructure with the following words:

> *"Critical infrastructures consist of those physical and information technology facilities, networks, services and assets which, if disrupted or destroyed, would have a serious impact on the health, safety, security or economic well-being of citizens or the effective functioning of governments in the Member States."* [3]

To reach the demands, the operators need to guarantee high availability of the network. A commonly used level to mark high availability is 99.999% [4], which means less than five and a half minutes of downtime over a year. Engineering for availability requires complex and redundant systems which can survive multiple types of unintended failures such as power outages and earthquakes, but it also needs to survive deliberate attacks against the systems availability such as remote network attacks. A network that is classified as a critical infrastructure has to be able to withstand deliberate attacks without causing unavailability to the network. Some types of attacks can be stopped by preventing intrusions with network firewalls and intrusion detection systems. However, software vulnerabilities may still exist and be exploited to cause unavailability. In order to prevent vulnerabilities that can be exploited by malicious input data, such input needs to be handled correctly by the application. Therefore, the system needs to be robust and tested thoroughly for possible vulnerabilities.

A successful attack on the telecom network may lead to a denial of service which will disturb end-users connectivity. Such a disturbance is not acceptable due to the very large number of users and critical services that would be affected. Emergency numbers would be affected and possibly unavailable over the telecom network, which could have negative consequences. These

circumstances put high demands on the system, both from end-users such as consumers and businesses as well as through government regulations.

The availability of telecom services is highly prioritized by telecom network providers and software testing has to be applied to ensure robustness of their systems. A robust system has the ability to handle malicious or unknown data without ending up with an unwanted behavior such as a system crash leading to a denial of service. Appropriate testing has to be applied to enhance the robustness of the system implementation. A well-known software testing technique for robustness testing is fuzzing, where malformed messages are injected into the system to test its implementation. The system is monitored for unwanted behavior to determine if the injected message caused any problem.

Third party robustness testing tools, as offered by Codenomicon [5] and P1 Security [6], have fully modeled protocols and advanced fuzzing algorithms to apply on their models. These testing tools have all the necessary means to perform robustness testing of various systems and implementations and are often adoptable to test not yet modeled protocols. However, one disadvantage of using third party tools is when internal information, e.g. source code, of the systems cannot be revealed. A black-box approach then has to be applied on the tested system which may have an influence on the fuzzers' efficiency. A fuzzing tool that utilizes the information of the implementation has all the means for having an increased performance. This can be knowledge of which message types that will be accepted and discarded upon arrival as well as code coverage issues. A white-box approach might be preferable in more complex systems, such as the systems of the telecom network.

Recent presentations [7, 8, 9, 10, 11] shows that fuzzing can find exploitable bugs in the second generation (2G) and third generation (3G) cellular network standards. This report will focus on the fourth generation (4G) cellular network standards and use knowledge of the system to reach deeper into the system implementation to perform directed and more sophisticated testing than what a third party test tool have access to.

1.1 Scope

The purpose of this thesis is to integrate robustness testing with an existing testing environment for functional and conformance testing. A comparison between TTCN-3 and GTT will be established to determine what testing enviroment to use. Robustness testing will be implemented with model-based fuzzing, using the models provided by the testing environment. The model-based fuzzer uses information from models, such as a network protocol, to intelligently generate data, such as protocol messages. The fuzzer will be general and applicable to different models, and tested on telecom equipment with models of a 3GPP protocol.

The telecom network will be tested due to its importance as a critical infrastructure. Fuzzing has proved itself useful in several previous works, where vulnerabilities are found and exploited. This thesis will be different from previous work due to its testing of real telecom network equipment in a controlled environment, with full control over the network. We will have access to the source code as well as experts in the field of telecom networks. This opens the possibility for efficient testing based on information only accessible from the inside.

1.2 Delimitations

- The proof of concept fuzzer will be specialized to work with an existing testing environment and the telecom protocols already tested there. It will not cover any other protocols or testing environments.

- After the study of the testing environments, only the chosen one, out of the available testing environments, will be used for the proof of concept fuzzer implementation.
- A fuzzing engine will either be developed from scratch or chosen from an open source third party tool. If a fuzzing engine is chosen, it will not be extended to include missing functionality of other fuzzing engines.
- If the fuzzer is capable of finding bugs, the source of the erroneous behavior will not be investigated. The developed fuzzer will only prove a concept that it is possible to find bugs, not troubleshoot the source of a potential error.

Chapter 2

Theory

This chapter provides the reader with detailed background information of the different topics covered in the thesis. The telecom network are described first with an overview of its history and details about the 4G standards. The two testing environments, TTCN-3 and GTT, of the target system are then described. Necessary information of the programming language Erlang are described, as well as the target systems hardware and software configuration. Lastly, the software testing technique called fuzzing is described.

2.1 Telecom networks

The telecom network is an interconnected network for long distance communication. It provides the means for voice and text communication, Internet access, and emergency services for end-users worldwide. An access point for wireless radio communication, called Radio Access Network (RAN), provides an interface to the telecom network. The RAN is connected to a core network that will handle all in- and outgoing traffic to a Public Switched Telephone Network (PSTN) for voice communication and to a Packet Data Networks (PDN), such as the Internet, private corporate, or service networks.

2.1.1 History

The different telecom network standards are divided into generations, where the first generation (1G) consisted of several different standards. These standards commonly used digital signaling but analog transmission of voice. One such 1G standard is the Nordic Mobile Telephone (NMT) which was used in parts of Europe, mainly in the north [12]. The competition of different standards mostly meant that a cell phone only worked with a specific network, and could not be used in any other network. Most 1G networks are today closed down.

The 2G standards sparked the transition from analog to digital transmission of voice. This also enabled the transmission of other data than voice, such as SMS text messages. The Global System for Mobile Communications (GSM) is such a 2G mobile communication standard and can today be accessed in over 219 countries worldwide [13] and have over 6.8 billion subscribers according to the International Telecommunications Union [1]. GSM supports voice and text communication as well as data transfers at low speeds. The GSM network is circuit switched, which means that there is a dedicated channel that transmits the voice in an ongoing call. The technology evolved over the years and with the addition of the General Packet Radio Service (GPRS), packet switching became available and with that came Internet access through the

telecom network. With the growth of Internet usage, an increasing demand for higher bandwidth came from end-users. A competing 2G standard is the IS-95 standard, marketed as cdmaOne, which is developed by Qualcomm [14]. Both GSM and IS-95 tried to meet the demands for higher bandwidth by launching new evolved standards.

The International Mobile Telecommunications-2000 (IMT-2000) is a specification by the ITU for standards that can support voice and high-speed data communication. Standards that comply with IMT-2000 are better known as the third Generation (3G) mobile communication systems [15]. Both the Third Generation Partnership Project (3GPP) and the Third Generation Partnership Project 2 (3GPP2) were founded based on the ITM-2000. The 3GPP developed Universal Mobile Telecommunications System (UMTS) as a successor to GSM [12]. The same core network, circuit switching and GPRS, is used in UMTS as in GSM. The radio interfaces for GSM and UMTS communication are different which means that two different RANs (Radio Access Networks) has to be used to support both standards. A competing 3G standard to UMTS is the CDMA2000 which is a development of cmdaOne by Qualcomm and the 3GPP2 [14]. Even with the 3G standards providing much higher bandwidth than the 2G standards, the demand for higher bandwidth is still increasing.

To further enhance packet switched communication with even higher speeds, the 4G communication standards has been developed. Initially there was several competing standards with the largest being 3GPP's Long Term Evolution (LTE), 3GPP2's Ultra Mobile Broadband (UMB) and IEEE 802.16, more commonly known as WiMAX. Common for all developed standards, and requirements of the ITU specification IMT-Advanced for 4G, is the path to a fully packet switched network [16]. The lead sponsor of UMB, Qualcomm, decided to end development of UMB and recommends LTE instead. The ITU later declared that two standards comply with their 4G definition IMT-Advanced: LTE-Advanced (LTE Release 10 and Beyond) and WirelessMAN-Advanced (Mobile WiMAX Release 2) [17]. With Qualcomm's recommendation of LTE, this is seen as the most natural upgrade path for current 3G network.

2.1.2 Evolved Packet System

The Evolved Packet System (EPS) provides the core network for LTE communication and is standardized by 3GPP. LTE is marketed as a 4G standard and the first available 4G services were launched in Stockholm, Sweden, and Oslo, Norway, on December 14, 2009 [18]. The

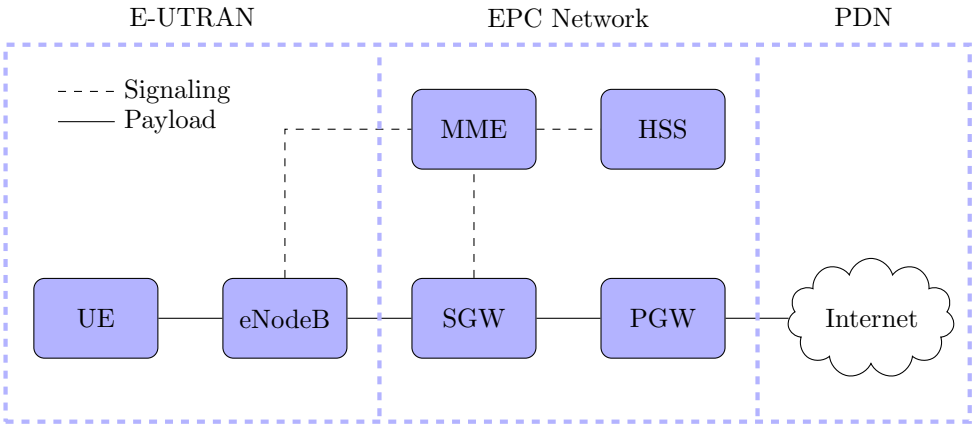


Figure 2.1: EPS network overview

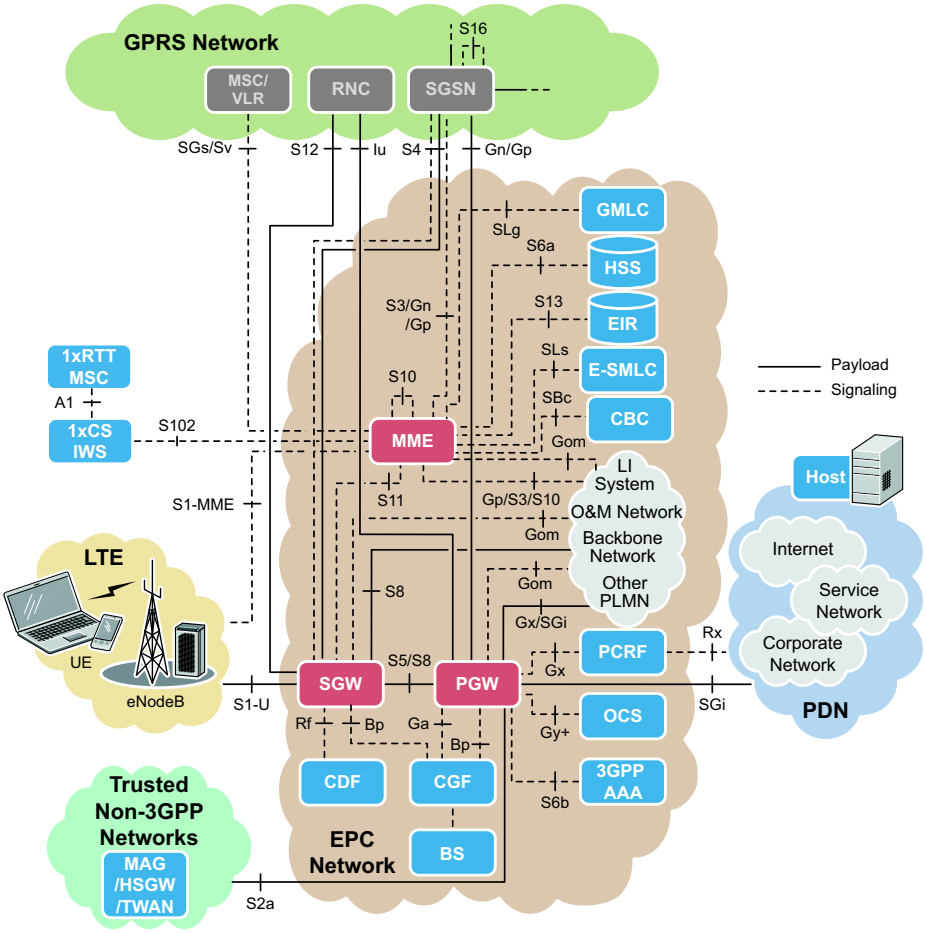


Figure 2.2: Overview of the EPS network [19]

EPS is divided into three main networks: Evolved Universal Terrestrial Radio Access Network (E-UTRAN), Evolved Packet Core (EPC) and PDN.

An overview of the EPS with the main components of each network can be seen in Figure 2.1. The E-UTRAN provides wireless access for User Equipment (UE), such as mobile phones and tablets, supporting LTE. The radio base station, in EPS called E-UTRAN Node B (eNodeB), provides signaling messages from the UE to the Mobility Management Entity (MME) and payload from the UE destined for a PDN.

The MME is a core node of the EPC network and is responsible for mobility handling and session management of a UE. The Home Subscriber Server (HSS) will provide the MME with authentication and authorization functionality of an attaching UE. When the HSS have authenticated the UE as a valid network user, a Serving Gateway (SGW) and a PDN Gateway (PGW) will be assigned to the UE. The SGW is the bridge between the EPC and the E-UTRAN and will route packets from the UE to the PGW and from the PGW to the E-UTRAN. The PGW is the bridge between the EPC network and the PDNs.

A more detailed illustration of the EPS network can be seen in Figure 2.2. Five different networks can be seen where the E-UTRAN (here called LTE), EPC network and the PDNs have

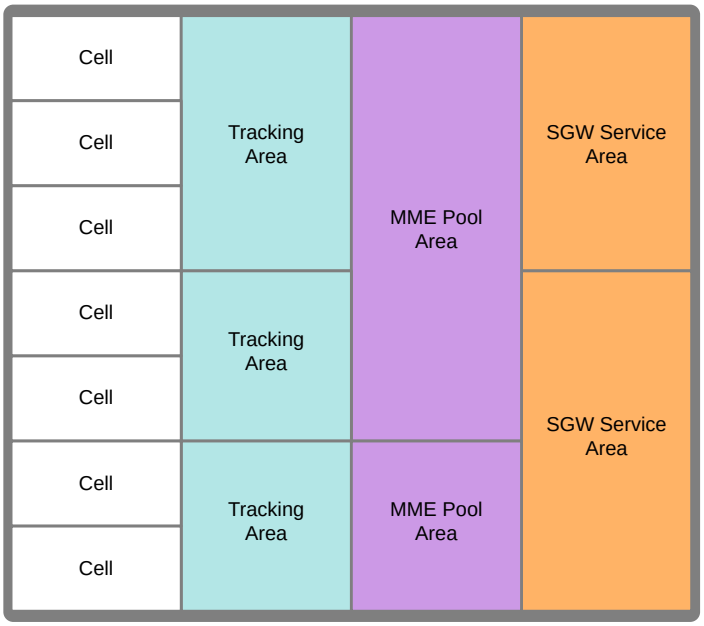


Figure 2.3: Network structure of the EPS radio network [19]

already been described. The Trusted Non-3GPP Networks can be seen in the lower left corner. These networks can be connected to make use of the EPC, but it is up to the operator to decide which of the networks standards that are allowed. The GPRS network, for 2G and 3G mobile communication, can be seen at the top of the figure. Also visible in the EPC network are several nodes which contribute in various ways. For example the Billing System (BS) node at the bottom of the EPC network illustration keeps track of subscriber usages to bill accordingly.

2.1.3 The Cellular Network

The RAN is divided into static geographical areas and the smallest geographical area in the EPC radio network is a cell. An example network structure of an EPC radio network can be seen in Figure 2.3. The size of a cell is based on the number of expected users in a specific geographical area and may vary in size. When a large number of users are expected, such as in a big city, a small geographical cell may be preferred, while a cell with a large geographical area may be more suitable on the country side. A cell is covered by a single eNodeB, but an eNodeB can cover multiple cells depending on the expected number of users and the capacity and range of the eNodeB.

As can be seen in Figure 2.3, a cell belongs to a Tracking Area (TA) and each TA contains at least one cell. The area covered by a TA can be from a small part of a city to an entire county. Each TA is covered by one MME, or several MME's if an MME pool configuration is used. If an MME pool is used the MME pool area defines the TAs a UE can move between without being required to reconnect to another MME. The MME Pool consists of one or several MMEs working in parallel within the MME pool area and providing load distribution, expansion possibilities and redundancy services. As long as a UE remains within the same MME pool serving area, it is attached to the same MME. If the MME is unavailable, the eNodeB will redirect the UE to another MME in the pool.

The last area in the cellular radio network of the EPC is the SGW serving area. This area defines for which cells a certain SGW can be used. The SGW serving area may not cover a whole MME pool area and if a UE moves outside the SGW serving area the MME has to redirect the UE to the dedicated SGW for the new cell.

2.1.4 Communication Interfaces

There are several communication interfaces in EPS to interconnect the nodes. Since different nodes in the EPC can be from different vendors, it is important to standardize the interfaces they communicate over. The EPS architecture with its interfaces is standardized by the 3GPP in detail in the standard TS 23.401 [20]. An overview of the interfaces with their name and connections can be seen in Figure 2.2. The protocols used in some important interfaces will be further described below and can be seen illustrated in Figure 2.4.

All UE communication is wireless and goes over the radio interface *Uu* with an eNodeB. The UE communicates with the eNodeB using the Radio Resource Control (RRC) protocol [21]. RRC encapsulates all different messages that is transferred between the network and the UE, both signaling and payload data.

The eNodeB has a group of interfaces towards the EPC network called the *S1* interfaces, consisting of the *S1-U* interface to an SGW and the *S1-MME* interface to an MME. The *S1-U* interface is based on Internet Protocol (IP) with User Datagram Protocol (UDP) as transport protocol, carrying GPRS Tunneling Protocol (GTP) User Plane version 1 (GTPv1-U) application data [22]. The GTPv1-U protocol carries encapsulated IP payload data between a UE and a PDN, such as the IP traffic when a user browses the web with a smartphone. The *S1-MME*

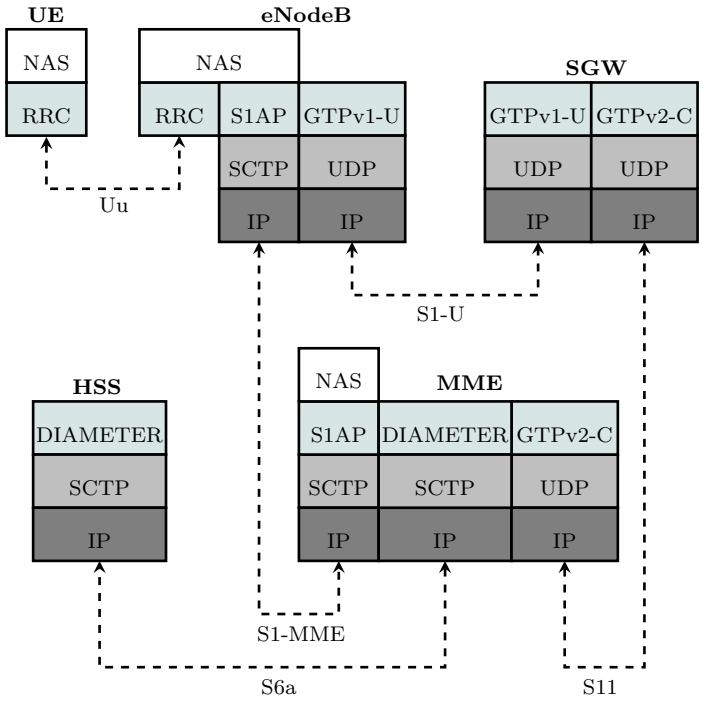


Figure 2.4: Subset of communication interfaces with their protocols

interface is IP based with Stream Control Transmission Protocol (SCTP) used as transport protocol. The S1 Application Protocol (S1AP) is used as the application protocol between the eNodeB and the MME, containing functions for configuring the signaling context for a UE and transferring messages between the MME and the UE. This communication between the MME and the UE over S1AP uses the Non-Access Stratum (NAS) protocol, which is further described in Section 2.2.

The MME communicates with several interfaces within the EPC network. Two selected interfaces are described here, out of importance to both the network and the thesis. The *S11* interface towards the SGW uses GTP Control Plane version 2 (GTPv2-C) over IP and UDP and carries signaling to setup and configure connections for a UE via the SGW [23]. The *S6a* interface towards the HSS carries the Diameter protocol over IP and SCTP to provide authentication and authorization of a UE and the subscriber [24, 25].

2.1.5 Mobility Management Entity

The MME is responsible for the attachment and detachment of a UE. It will select an SGW and a PGW for the UE to use in order to connect to a PDN. The MME will also assure the sustainability of this connection from the UE to the PDN network over time. In the attachment process of a UE, the MME performs an authentication of the UE to determine if the UE has the correct access rights to use the EPS. This authentication is done in conjunction with the HSS which contains subscription information for authentication and authorization of subscribers.

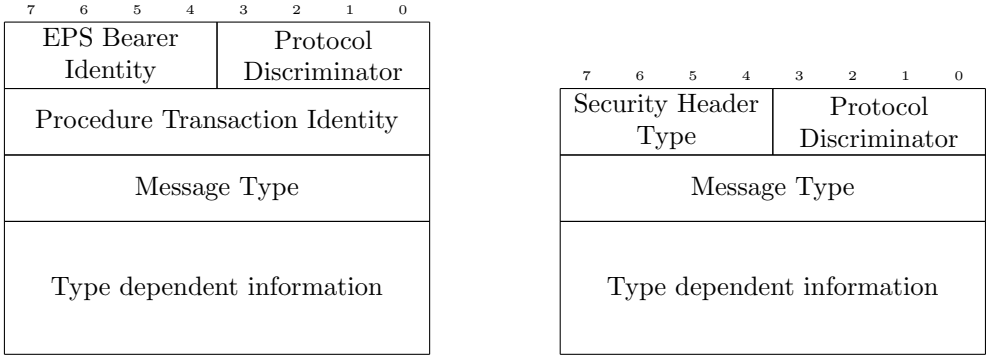
A UE that is moved into a new TA has to establish a new connection to the eNodeB that covers the entered cell. The MME will manage the handover to the new eNodeB and make sure the connection to the PDN network is sustained. If a UE enters a cell in the EPC radio network, the MME will also take care of the procedure of transferring a connected UE of a 2G or 3G network to the LTE network that is currently available and the other way around. Communication between the MME and the UE is transferred with the Non-Access Stratum (NAS) application layer protocol described in more detail in the next section.

2.2 Non-Access Stratum Protocol

The NAS protocol is used for signaling between the UE and the MME [26]. NAS messages are transported over RRC between the UE and the eNodeB, and forwarded unmodified by the eNodeB to the MME via the S1-MME interface, encapsulated in S1AP messages as described in Section 2.1.4. The protocol is internally a set of two protocols which have different responsibilities; mobility management by the EPS Mobility Management (EMM) protocol and session management by the EPS Session Management (ESM) protocol.

The EMM part of NAS is responsible for mobility related procedures in the E-UTRAN, authentication of UEs and security related issues. The 3GPP specification has separated the functionality of the EMM protocol into three sections. The first is the EMM common procedures where all network-initiated mechanisms are described such as authentication requests and encryption initialization. The second section covers the EMM specific procedures that identify all UE-initiated procedures. Here we find procedures such as attach and detach to and from the EPC. The third and last section is the EPS Connection Management (ECM) which provides means to maintain the connection to the EPC.

The ESM part of NAS offers functionality for establishment and handling of user data. To let the UE communicate with a PDN, the ESM is used to establish a PDN connection. Within this connection, one or several EPS bearers exist which are logical connections carrying user data [20]. All traffic over a PDN connection are assigned to and carried over a specific bearer.



(a) ESM message **(b)** EMM message

Figure 2.5: NAS plain structure

```
1  template NAS_Message example_esm :=
2  {
3      ProtocolDiscriminator := '0010'B, // ESM Message
4      EPS_messages := {
5          EPS_SessionManagement := {
6              EPSBearerIdentity := '0111'B,
7              ProcedureTransactionIdentifier := '00001111'B,
8              MessageType := '11010010'B, // PDN_DisconnectReject
9              MessageTypeData := {
10                 PDN_DisconnectReject := {
11                     // Type dependent information
12                 }
13             }
14         }
15     }
16 }
```

Figure 2.6: Example NAS ESM message described in TTCN-3

There is always at least one bearer, the default bearer, in every PDN connection. Other bearers are called dedicated bearers and are setup with the ESM protocol. These dedicated bearers are configured with a packet filter, to specify the packets that should belong to a bearer. They can also be configured with different quality of service (QoS) specifications, for example to prioritize a bearer carrying voice data. The default bearer carries all traffic that is not matched by any dedicated bearers filter.

2.2.1 Protocol Structure

The EMM and ESM protocol structure is illustrated in Figure 2.5. The four bit *Protocol Discriminator* field defines the NAS message structure. For example, a *Protocol Discriminator* with the bit sequence 0010 is defined as an ESM message while the bit sequence 0111 is an EMM message [27]. The *EPS Bearer Identity* field in ESM is used to identify the message flow and the *Security Header Type* field in EMM contains information regarding the security protection of

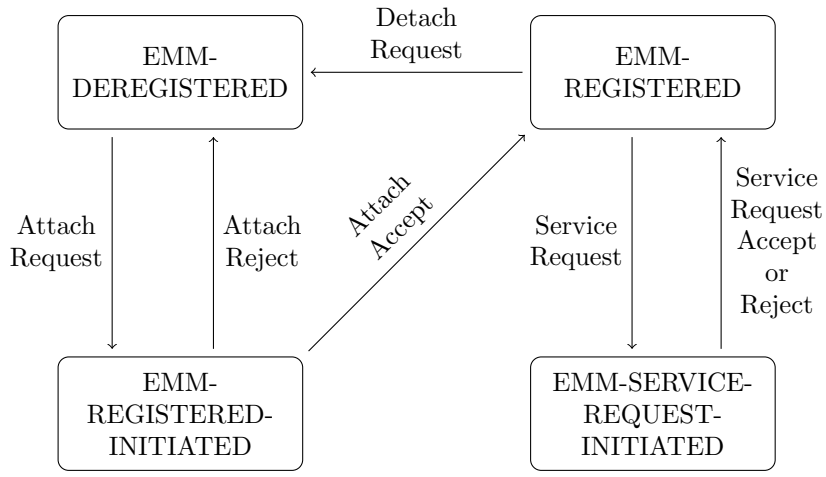


Figure 2.7: Simplified EMM state machine for the MME

the NAS message. The ESM message has an extra octet for the *Procedure Transaction Identity* field which allows 254 bi-directional transactions and two reserved values. The *Message Type* field defines the message type of ESM and EMM respectively. The last field is dependent on the message type and may differ in length and content.

An example template of an ESM message can be seen in Figure 2.6, defined in the TTCN-3 language. The protocol discriminator can be seen on line 3 with the value defined for ESM, 0010. This is the reason why the EPS Message contains an ESM message in which the three fields for an ESM message is defined. The four bit *EPS Bearer Identity* on line 6, the eight bit *Procedure Transaction Identity* on line 7, and the eight bit *Message Type* on line 8 are all mandatory fields for an ESM message. The message type data field contains the ESM message defined by the *Message Type*, which in this case is a *PDN Disconnect Reject* message. Values in this field are dependent on the message type and will be filled with data according to the message type specifications. The *PDN Disconnect Reject* message contains no data, which means that the *Type Dependent Information* field will be of zero bit size.

2.2.2 Encapsulation

NAS messages are sent from the UE to the eNodeB via the RRC protocol. The eNodeB communicates with the MME over the S1AP protocol where NAS messages from the UE are encapsulated unaltered for transfer to the MME. There are three ways to transport NAS messages within S1AP, either as an *Initial Message*, an *Uplink NAS Transport*, or a *Downlink NAS Transport* [28]. The *Initial Message* type is used when the first NAS message to be forwarded to the MME arrives at the eNodeB from the UE. The eNodeB establishes a logical S1-connection between the UE and the MME by generating a new and unique eNB-UE-S1AP ID and includes this in the message along with the NAS message. When the connection exists, the eNodeB will send the NAS messages as a *Uplink NAS Transport* message, which consists of the ID of the S1-connection and the NAS message. As soon as the connection is closed, a new one has to be opened by sending the initial message as described above.

Table 2.1: EMM sublayer states

• EMM sublayer states in the UE	EMM sublayer states in the MME
• EMM-NULL • EMM-DEREGISTERED • EMM-DEREGISTERED-INITIATED • EMM-REGISTERED-INITIATED • EMM-REGISTERED • EMM-TRACKING-AREA-UPDATE-INITIATED • EMM-SERVICE-REQUEST-INITIATED	• EMM-DEREGISTERED • EMM-DEREGISTERED-INITIATED • EMM-COMMON-PROCEDURE-INITIATED • EMM-REGISTERED

Table 2.2: ESM sublayer states

ESM sublayer states in the UE	ESM sublayer states in the MME
• BEARER-CONTEXT-INACTIVE • BEARER-CONTEXT-ACTIVE • PROCEDURE-TRANSACTION-INACTIVE • PROCEDURE-TRANSACTION-PENDING	• BEARER-CONTEXT-INACTIVE • BEARER-CONTEXT-INACTIVE-PENDING • BEARER-CONTEXT-ACTIVE • BEARER-CONTEXT-ACTIVE-PENDING • BEARER-CONTEXT-MODIFY-PENDING • PROCEDURE-TRANSACTION-INACTIVE • PROCEDURE-TRANSACTION-PENDING

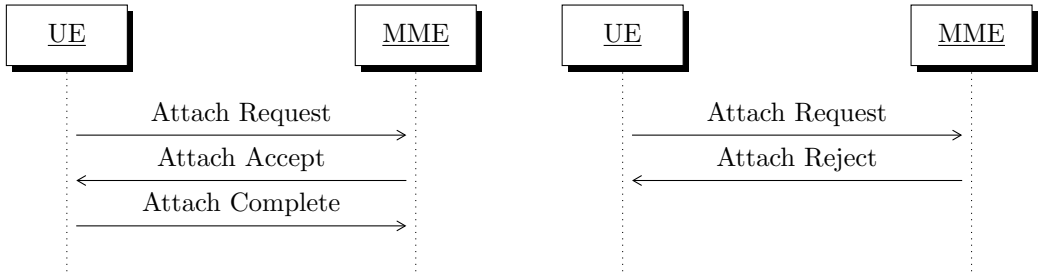
2.2.3 State Machines

Four different state machines are present in the system to determine the connection state of an MME and a UE. Two state machines for the communication with the E-UTRAN over the RRC respectively S1AP protocol and two for the UE and MME over the NAS protocol.

The UE and the MME will both establish a connection to the E-UTRAN where both the UE and MME can be in either *EMM-IDLE* or *EMM-CONNECTED* mode [26]. The UE and the MME communicates with the E-UTRAN with separate interfaces. The UE uses radio communication to establish a connection to the E-UTRAN over the RRC layer in the protocol stack while the MME uses the S1AP layer with a wired connection in the S1-MME interface. The mode will shift from *EMM-IDLE* to *EMM-CONNECTED* when an RRC respectively S1AP connection has been established. The mode will shift back when this connection is released.

The EMM part of NAS is built around two state machines which exist for every UE; one state machine in the UE and one in the MME. The UE state machine starts in the state *EMM-NULL* and needs to be in state *EMM-REGISTERED* to perform most EMM operations. The main EMM states for which the UE keeps track of can be seen in Table 2.1 where *EMM-REGISTERED* and *EMM-DEREGISTERED* have substates of their own, but these will not be presented in this table. In order to get to *EMM-REGISTERED* an attach request to the network has to be achieved. The attach procedure will establish an EMM context, containing information to keep the connection alive.

The EMM state machine for the MME has only four main states which are stated in Table 2.1. A simplified state machine with the most important states, for the understanding of this master thesis report, and with the messages that change the states are shown in Figure 2.7. The MME state machine will begin in the state *EMM-DEREGISTERED* where the UE is detached and the MME have no EMM context. As stated previously, an attach procedure has to be achieved in order to change the state to *EMM-REGISTERED*.



(a) Successful attach **(b)** Unsuccessful attach

Figure 2.8: NAS attach procedure over logical S1 connection

The main function of the ESM sublayer is to provide support for EPS bearer context handling for both the UE and the MME. It provides the means for activation, deactivation and modification of EPS bearer contexts. ESM has one state machine for every bearer context and for both the UE and the MME. These state machines in the UE and the MME have four respectively six main states which can be seen in Table 2.2. The states indicate if a bearer is active, inactive, or in a modification transaction.

2.2.4 Attach and Service Request

A UE that wants to use the EPS must announce its presence in the network. This is achieved via a UE initiated attach procedure to the MME which starts with the UE sending an *Attach Request* message. An illustration of the attach procedure can be seen in Figure 2.8. The MME will handle the request and determine if an attach procedure to the network is permitted. A couple of procedures have to be established in order for a UE to be successfully attached. The subscriber is authenticated, a session to the SGW is created, bearers are established, and contexts for the communication are set up. If any complication occurs during the attach procedure an *Attach Reject* message will be returned to the UE which have to restart the attach procedure to get access to the network. If successful, an *Attach Accept* message is returned from the MME instead. The UE will confirm the attach with an *Attach Complete* message and enter the *EMM-REGISTERED* state.

When the UE is in state *EMM-REGISTERED* and *EMM-IDLE* mode, no signaling connection is active and the only allowed messages to send are *Attach Request*, *Detach Request*, *Tracking Area Update Request*, *Service Request*, and *Extended Service Request*. These are sent as initial NAS messages by the eNodeB to setup a connection to the MME for the UE. When the initial NAS message has been sent and an NAS message from the MME is received, the UE enters *EMM-CONNECTED* mode. As mentioned earlier, one of the initial messages is *Service Request* which is very commonly used when a UE wants to send signaling data to the network [26]. An accepted *Service Request* message will be answered with a *Service Accept*, and a denied request will be answered with a *Service Reject*.

2.3 Testing and Test Control Notation Version 3

This section will introduce the basic structure of the testing language TTCN-3 and the associative entities for each interface as well as the test executable that uses the interfaces to communicate with the system under test (SUT).

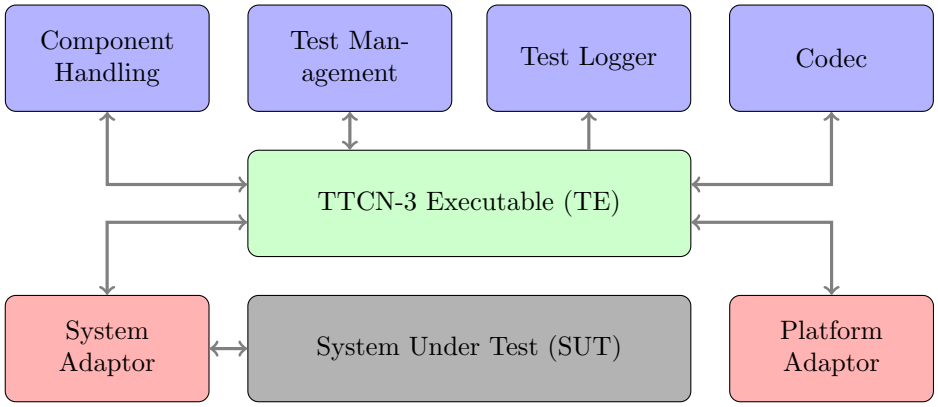


Figure 2.9: TTCN-3 Test System

TTCN-3 is a standardized testing language and environment developed and maintained by the European Telecommunications Standards Institute (ETSI) [29]. It is designed purely for testing and specially for black-box testing. The TTCN-3 language has seven basic built-in types: `integer`, `float`, `boolean`, `bitstring`, `hexstring`, `octetstring`, and `charstring` [30]. The types correspond to what the names are normally used for, with `integer` having integer values and such. The types are also of no fixed size, with the ability to create a string type such as `bitstring` with any length (e.g. 100 bits). There are also special types such as `verdicttype` which is used to set a verdict of a test case (e.g. pass or fail). It is possible to define subtypes to the built-in types and introduce restrictions to them, such as restricting a `bitstring` to a length of four bits. This subtype can then be used in the same way as the other types.

The language also provides the possibility to define structured types [30]. A `record` structure will define a new type with typed and named fields. When creating an instance of such a structure, these fields have to be instantiated with the correct type of the field. The fields can also be marked as optional and then omitted when creating the structure. A `union` structure will also define a new type with typed and named fields. The difference compared to `record` is that a `union` can only hold one value, which means that only one of the fields can be chosen. There are several other structured types, such as `set` and `enumerate`.

In order to use the TTCN-3 language to test network protocols, the protocols are modeled using the language types and structures. These models are based on the protocol specification or standard, which states the different fields and length of the protocol messages. Such specifications can be e.g. Request for Comments (RFC) for internet protocols or 3GPP standards for telecom protocols. A test case is then created containing sequential ordered operations that should be performed, such as creating a message using the models, sending and receiving messages to and from an SUT respectively. The test cases combined with the models are compiled into a TTCN-3 executable (TE). A test system provides the TE with all necessary means for communication between the TE and the SUT as well as the user of the test system and the TE. An overview of the TTCN-3 test system is illustrated in Figure 2.9.

The TTCN-3 test system has two standardized interfaces: The TTCN-3 Control Interface [31] and the TTCN-3 Runtime Interface [32]. The TTCN-3 Control Interface provides communication from and to the TE for the test system user. It also defines modules for test distribution and encoding/decoding of messages. The four modules defined in TCI are: test management, test component handler, logging, and codecs. The TTCN-3 Runtime Interface has two modules that define the communication between the TE and the SUT as well as the TE and the platform.

The two modules in the TTCN-3 Runtime Interface are: SUT Adaptor and Platform Adaptor.

2.3.1 TTCN-3 Control Interface

The TTCN-3 Control Interface is responsible for test execution management, distribution of test execution over different test devices, encoding/decoding of messages to and from the SUT, and logging information of the test execution. These tasks are divided over four different entities: Test Management, Component Handler, Test Logging, and Codecs.

The Test Management entity is responsible for all management of the test system. First, an initialization of the test system is performed. When this is done the test execution is started within the Test Management entity. The Test Management entity is also responsible for all invocations of TTCN-3 modules, i.e. propagation of module parameters to the test execution.

The Component Handler entity makes it possible to distribute the test executor over several nodes. It provides the means for communication for the distributed test system entities. By this, synchronization of the distributed nodes can be established.

The Test Logging entity provides the information of the execution. It performs test event logging and presentation of all events during execution. The log can contain information such as which test component that have been created, what data was sent to the SUT and when, what timers that have been started, stopped, or timed out etc.

The codecs are responsible for encoding and decoding the TTCN-3 values to and from the communication data format respectively. This format is used in communication with the SUT and often consist of transforming the data to and from binary strings. The codec for a certain TTCN-3 type defines how the values should be represented in these binary strings, e.g. in what order they should appear and the bit endianness.

2.3.2 TTCN-3 Runtime Interface

The TTCN-3 Runtime Interface handles all communication between the TE and the SUT. All timers and external functions are defined and called within this interface. The TTCN-3 Runtime Interface standard provides information about two entities: SUT Adaptor and Platform Adaptor [32].

The SUT Adaptor implements the real communication link between the test executable and the SUT, and forwards events to each end of the link. The SUT Adaptor takes the send or receive requests from the TE and sends messages on the link or to the TE respectively.

The Platform Adaptor implements all external functions and timers. All external functions that are called from the TE go through this interface. The Platform Adaptor takes care of all interactions with the timers, such as starting, stopping, and reading.

2.3.3 TITAN: Test Execution Environment

TITAN is an Ericsson developed test compiler and executor for TTCN-3 test suites. TITAN was initially developed during a Master of Science thesis at Ericsson in the beginning of year 2000 [33]. More functionality has been added since the first prototype and TITAN is now a complete compiler and test execution environment for TTCN-3 test cases with support for parallel distribution and ASN.1 modules [34].

TITAN uses C++ as an intermediate language before the compilation of an executable file. TITAN takes TTCN-3 and ASN.1 modules as input to generate C++ code. The generated C++ is compiled together with a base library and test ports to a TE, in TITAN called Executable Test Suite (ETS). The ETS contains all necessary means for test execution, including logging

capabilities, coding and decoding functionality and includes procedures for communication with the SUT as well as the test system user. TITAN also provides a complete test environment for configuration and execution of an ETS.

TITAN does not follow the TTCN-3 Runtime Interface and TTCN-3 Control Interface standards introduced by ETSI. In 2002-2003 when ETSI released the two standards, TITAN already had a complete test system with its own proprietary interfaces [33]. The TITAN interfaces are similar to those standardized by ETSI but not fully compliant for various reasons, such as efficiency. An advantage with the TITAN SUT adapter is that a port instantiation only handles one TTCN-3 communication port which corresponds to one protocol. In this way distribution of messages of various protocols is solved by the interface, as opposed to the TTCN-3 Runtime Interface where all messages will be handled by the same function of a single adapter.

TITAN has a "Negative testing mode" feature which gives the ability to write TTCN-3 test cases where values in messages are overridden. This makes it possible to use fault injection in the TTCN-3 environment. The values are statically assigned at compile time and are capable of overriding arbitrary fields of a message.

2.4 Erlang

Erlang is an open source functional programming language, developed by Ericsson in 1986 [35]. It was developed for the telecom network to be able to handle a large amount of concurrent processes. An Erlang process is implemented as a lightweight process where no shared memory is used [36]. The independence of each process means that if a process crashes, the other processes will not be affected. This approach is desired in a network with a high number of connected entities. The philosophy of Erlang is to handle faults by letting a process crash and restart it.

(a) Restart one strategy where process 2 crashes **(b)** Restart all strategy where process 2 crashes

Figure 2.10: Erlang supervisor tree restart strategies

A basic concept in Erlang is a supervision tree with two different processes: workers and supervisors [37]. The worker is a process that executes some type of calculation, while the supervisor is a process that monitors the worker. A supervisor can supervise both workers and other supervisors, but a worker cannot have a child of its own. Two examples of Erlang supervisor trees are illustrated in Figure 2.10. When a worker crashes, it is the supervisor's responsibility to restart it again. The supervisor can be configured to restart the worker that crashed or an arbitrary number of process children depending on the system configuration. The supervisor is configured with a restart strategy which defines the actions to take after a child crashes. Examples of restart strategies are to only restart the crashed child (Figure 2.10a), restart all children with a higher id than the crashed child, or restart all child processes (Figure 2.10b).

A maximum restart frequency is defined with two variables: `MaxR` and `MaxT`. If the number of restarts exceeds `MaxR` during the time interval `MaxT`, then the supervisor will crash, including crashing all its children, and the supervisor above will take appropriate actions according to its restart strategy.

The supervisor tree can be of different levels depending on system architecture and each level is dependent on `MaxR` and `MaxT` values for the restart procedure. In this way, different parts of a system can be restarted without affecting other parts of the tree. This method of process handling is particularly effective in systems where one process handles one task, such as a connection to a device, where a crash of a process will not affect other processes in the same tree. However, frequent crashes in one part of the system may escalate to force restarts of correct processes in other parts of the tree. The intention of escalation restarts is to prevent a situation where a child process crashes repeatedly for the same reason. If child processes are crashing repeatedly, the reason for the crashes is believed to be solvable by restarting a larger part of the system. If this does not work, a greater part of the system is restarted until the problem is solved.

2.5 Ericsson SGSN-MME

The Serving GPRS Support Node - Mobility Management Entity (SGSN-MME) developed by Ericsson provides SGSN functionality for GSM (2G) and WCDMA (3G) network access and MME functionality for LTE (4G) network access which includes the EPC network. A network with a functional SGSN for GSM and WCDMA access can be upgraded to an SGSN-MME to support triple 3GPP access (GSM, WCDMA, and LTE) with a simple software upgrade [38]. The SGSN-MME can also be configured with dual access for either GSM and WCDMA or WCDMA and LTE according to the customers' desires. Ericsson SGSN-MME offers scalability by deploying a pooled configuration where one node supports 18 million subscribers and up to 64 nodes can be pooled in the same configuration which gives support for 1152 million subscribers.

2.5.1 Hardware

Ericsson SGSN-MME is held within a cabinet that can hold up to three magazines which each contains various number of Plug-In Units (PIU), depending on the hardware configuration. The PIUs are interconnected through the backplane of a magazine which provides dual redundant power distribution and dual redundant Ethernet connectivity to all PIU slots. The latest hardware (MkVIII) has three different PIUs:

- System Control Switch Board version 2 (SCXB2),
- Component Main Switch Board version 3 (CMXB3),
- Generic Ericsson Processor board version 3 (GEP3).

The SCXB2 handles Ethernet layer 2 and layer 3 for internal switching and external IP interfaces. The CMXB3 handles Ethernet layer 2 switching for communication between the magazines. The GEP3 can be configured to take different roles and is available in two different versions, with or without storage media. The GEP3 with storage media takes the role as a File Server Board (FSB). The GEP3 FSB is available with either a mechanical hard disk or a Solid State Disk (SSD). A GEP3 without storage capabilities can take several roles:

- Application Processor (AP)
- Device Processor (DP)
- Node Control Board (NCB)

The AP card handles traffic control activities such as mobility and session management and high-level protocol processing. The AP card will handle connections to subscribers and internal system activities such as recovery and distribution. The DP card will manage the internal distribution of SCTP messages. The NCB provides central support and functions for the boards in the magazine, such as hardware and software monitoring and distribution, and SGSN-MME supervision.

A typical hardware setup for a magazine would be to have two SCXB2, CMXB3 and NCB respectively. The remaining slots would either be AP boards, DP boards or FSBs depending on the purpose of the hardware configuration.

2.5.2 Software

The SGSN-MME applications handling high-level protocols are largely implemented in the Erlang programming language. As described earlier in Section 2.4, using a supervisor tree with workers is recommended when designing software in Erlang and worker processes are supposed to crash upon failure instead of recovering. The software design of the MME follows the Erlang philosophy with supervisor trees, where every attached UE is handled by a worker process. Upon any unexpected failure occurring while handling traffic from a user, the worker will crash which results in the UE connection handled by that worker being dropped. This way, only that single UE is affected by the failure while the other connected UEs are left unaffected working as usual.

The Erlang supervisor tree of the MME contains supervisors as nodes and static and dynamic workers as leaves. Dynamically spawned worker processes (dynamic workers) are located at the bottom of the supervisor tree structure. A typical assignment of a dynamic worker process would be to handle a connection of a single UE. The process will be terminated either when the UE disconnects from the network, or when an error occurs which causes it to crash.

2.6 GSN Test Tool

The GSN Test Tool (GTT) is an Ericsson specific test framework for testing SGSN-MME functionality. The GTT framework has support for simulating all nodes which communicates with the SGSN-MME and can be run towards a simulated or a real SGSN-MME node. Test cases are written in GTT which is an Erlang based framework. A test case sends a message to the SGSN-MME and verifies the correctness of the received answer. Verification of returned data and observations of state changes in the SGSN-MME is used to distinguish the outcome of the test case which could either be pass or fail.

The GTT framework runs on a simulated version of an SGSN-MME node were the communication links is replaced by a shared memory network. The shared memory network has support for all interfaces to the node. The operating systems run on several virtual Linux and Solaris machines. The applications used in the SGSN-MME are however the same as in a real node.

GTT is used for traffic verification and has state awareness of the SGSN-MME to find errors in the system implementation at an early stage in the development process. It has support for full test execution automation with integrated functionality for result and log presentation. The logs will provide the designer with the information needed to find the source of the error.

2.7 Fuzzing

Fuzzing is a software testing technique where random or semi-random data is injected into an application or system which is observed for faulty or unwanted behavior to distinguish the mes-

sage that caused the fault [39]. Fuzzing is often used for security related issues such as finding system implementation flaws to increase system robustness. A robust system is capable of handling manipulated or random input data without disturbing legitimate system users. Systems with a large number of users have demands on reliability and availability and have to be engineered for robustness to withstand possible attacks, especially in a critical infrastructure such as the telecom network. The biggest field of application of fuzzing is security related testing for vulnerability and reliability issues to improve robustness.

2.7.1 History

Software testing with automatic random test case generation to stress the system implementation dates back to 1970 when K. V. Hanford introduced the "syntax machine" [40]. The "syntax machine" generated random test cases for any suitable programming language. The test cases were syntactically correct, but the results they produced were unpredictable and uncheckable. However, the number of test cases which could be generated by the "syntax machine" made it a valuable tool for reliability testing.

The term "fuzzer" originates from the University of Wisconsin by Barton Miller in 1988 [41], almost two decades later. It was one of the subjects which could be selected during a class project, taught by Barton Miller, where the students where supposed to develop a basic command-line fuzzer to test the robustness of Unix programs by sending randomly generated data in the intention to make them crash. The developed class project fuzzer was able to generate a random byte stream that was used as input to `stdin` via a pipe. The SUT was monitored to determine when the execution terminated for each input stream. If an error had occurred the application would generate a `core` file in the file system. If such file was found it was concluded that the random input stream crashed the application [42].

2.7.2 Black- and White-Box Testing

Software testing of applications and systems implementation can be established with two different approaches. Either the system is seen as a closed box, black-box, which given some input will produce some output, or it can be seen as an open box, white-box, where the internal structure is known and can be utilized to create test cases.

An advantage with black-box testing is that no internal structure or specific knowledge of the system is generally required [43]. The developer though has to be aware of what the SUT is supposed to achieve but exactly how this is done is not important. This is a commonly used method for fuzzing because of its simplicity, but it might not be suitable in more complex systems, such as where a state machine is present. The state machine may cause a big amount of the fuzzed messages to be discarded and never reach the internal structure of the SUT that the tester wants to test. This also implies that certain code coverage cannot be guaranteed because the system is treated as a black-box. It will not know how the messages can be generated to cover the parts that not yet have been tested.

In a more complex system, a white-box testing approach might be preferred if possible. White-box testing utilizes information of the SUTs internal structure to generate messages [43]. The aim of white-box testing is to expand the attack surface by intentionally produce messages that will hit certain paths in the source code. Paths that have been created by decisions based functions such as if- or switch-statements that is input dependent. The fuzzer can be directed to fuzz a specific part of the system by sending a message with certain static fields or a sequence of legitimate messages to reach a wanted state before the malformed messages are sent. The tester have to utilize the knowledge of the SUT to increase the efficiency of the fuzzer by not generating

messages that will be discarded or are already known to generate an erroneous behavior.

2.7.3 Mutation and Generation-based Fuzzing

There are no strict standards to follow when developing a fuzzer. As the authors point out in [44], fuzzing has no precise rules and its efficiency depends on the creativity of the developer. However, fuzzing can be divided into two distinct procedures [45]: mutation-based or generation-based fuzzing.

Mutation-based fuzzing needs to extract valid messages of the targeting protocol to be able to create fuzzed messages [45]. A mutation-based message can be seen in Figure 2.11, where the original hex string has been mutated in four different byte fields. Malformed messages are created from legitimate messages by applying various mutation rules. The rules are based on the mutation property, such as adding, removing, substituting, or bit flipping of data, and the data it should be applied on, such as protocol fields, sequences, bytes, or bits. The mutation properties can be applied on an arbitrary number of data segments. The mutation rules can be statically created or dynamically generated by random selection.

Original	46	75	7A	7A	69	6E	67	21	3F
Mutated	46	75	7A	7A	**65**	**64**	67	**67**	**7F**

Figure 2.11: Mutation of message

Generation-based fuzzing builds malformed messages from scratch. To achieve this, protocol specifications are needed. The messages are generated according to the types specified in the specifications. An example can be seen in Figure 2.12 where a message has been generated from a protocol specification. Some fields have to be statically assigned for communication means. This can be checksums or other control fields that cause the system to discard them at arrival. The aim of generation-based fuzzing is to generate semi-random messages where control fields are statically set to not disturb communication and to randomly generate all other data according to the protocol specifications.

Specifications	STRING	INTEGER	BOOLEAN
Generated	“Testing”	12345	TRUE

Figure 2.12: Generation of message

2.7.4 Fuzzing Techniques

What fuzzing technique to use for a specific SUT may vary depending on the knowledge and complexity of the SUT. With no knowledge of the system, also known as black-box testing, a random-based fuzzer may be suitable while in a more complex system an intelligent fuzzer, white-box fuzzing, may have to be considered in order to reach deeper into the system implementation before sending malformed messages. To describe the level of system awareness the terms “dumb” and “intelligent” fuzzing are frequently used by several authors. Dumb fuzzing refers to a limited awareness of the protocol and system implementation and a more intelligent fuzzer may have all, or close to all, knowledge. A dumb fuzzer is easy to construct but inefficient while the intelligent fuzzer is striving for a higher efficiency while having longer construction time. This section will

describe different fuzzing techniques, starting with a dumb fuzzer and ending up with a more intelligent fuzzing procedure.

A random-based fuzzer has no awareness, or limited awareness, of the message structure. It is a "dumb" fuzzing method that uses a black-box testing approach often with a mutation-based message generation. Random-based fuzzers often operate with functions such as replacing, adding and removing bits or bytes in a legit message. One approach to implement a random-fuzzer can be to listen to the network and catch a valid message sent to the SUT. One or several functions named above are then applied to the message. The outcome is a malformed message that is sent to the SUT which is monitored for unwanted behavior. This procedure is repeated until erroneous behavior is detected. Though, due to lack of knowledge of the message structure, random-based fuzzers may have problems fuzzing other messages than the first message in a complex protocol where a sequence of messages has to be sent to reach a specific state of the SUT.

A fuzzing technique which is more "intelligent" than the random-based fuzzer is a block-based fuzzer. The idea of a block-based fuzzer is to break down the protocol into two different block styles: a static block, that will remain the same in all fuzzed messages, and a variable block, where the fuzzing will occur. The variable block may be tagged with different types such as integer or string and is fuzzed according to the type.

A model-based fuzzer has knowledge of the protocol specification from which a model of the protocol is generated and executed. It is often used with a white-box generation-based testing method and is the most "intelligent" of the described fuzzing approaches. Complex interactions with the SUT can be accomplished with greater code coverage which implies that fewer test cases will cover a greater part of the implementation [46]. Different states can be reached before fuzzed malformed messages are delivered to the SUT, thus increasing the possibility to find unwanted system behaviors. By utilizing the protocol specifications, test cases can be targeted at certain parts of the system. A model-based fuzzer is more complex than other fuzzing techniques, but the time spent with design and implementation can decrease the execution time of the fuzzer gaining a significant advantage if the fuzzer is frequently used.

Chapter 3

Related Work

Previous work related to fuzzing, telecom and the combination of the two are interesting to this thesis. To show the importance of robustness in a telecom network, a series of attacks related to telecom are presented. These attacks show that the telecom devices and networks have been sensitive to attacks before, and probably still are. Many of the attacks also make use of fuzzing to find vulnerabilities, which shows that this method is useful in this area as well.

Since telecom networks has often been seen as protected with its secrecy and regulations, input from UEs could be regarded as more trusted than in other networks. This means that malicious input from attackers was seen as impossible or unlikely. However, several open source projects now show that the network is being tested with open tools freely available for anyone. The LTE standard is much more open than the GSM standard was and therefore should open implementations be easier without the need of reverse engineering. While we present a tool which handles the GSM stack very well, we also see tendencies of the open source community targeting LTE. With open tools for communicating with the network, an attacker can focus on the actual attack and not on reverse engineering the network stack. It also means that untested and uncertified tools that may behave incorrectly connect to the networks, making it even more important to handle inconsistencies.

This thesis aims to design a robustness testing tool to proactively find vulnerabilities and weaknesses before being exploited by a malicious user. The tool is focused on integrating with a conformance test environment, and previous attempts to do this are presented here. The main focus has been to investigate fuzzing within a TTCN-3 environment since it is a standardized testing environment and very well used in the industry. There was also previous research in this field that was very close to the goals of this thesis.

While presenting research, several fuzzing tools are also investigated. Both open source variants with different specialties as well as commercial tools with support for telecom related protocols. The tools investigated were preferably model-based fuzzers, since that was one of the main aims of the thesis. Several differences between the investigated fuzzers exist and are presented here.

3.1 Telecom Attacks

Recent studies [7, 8, 9, 10] points out the existence of vulnerabilities in the cellular network. Some of the studies have put their focus on the SMS implementation, as the study of Mulliner et al. [8].

Mulliner et al. make the use of fuzzing techniques in their paper [8] to generate, what they call, an SMS of death. The aim of their paper is to find vulnerabilities in the SMS implementation with an open source library for SMS generation written in Python. The equipment at hand was a GSM base station connected to a laptop running the open source base station controller (OpenBSC) configured as desired for the experiment, as well as a couple of mobile phones from the leading telephone companies. Their methodology was as follows: generate a database of fuzzed SMS messages, set up a communication channel between the cell phone and the base station, send SMS message, and close the channel. The cell phone was monitored for crashes and the messages that generated a crash were flagged as an invalid message. The authors found malformed SMSes that disconnected the phone from the network and some even provoked a reboot of the phone, showing that fuzzing is a valuable tool to find vulnerabilities in the SMS implementation of numerous leading brands. By making use of the network automatically retransmitting SMSes that wasn't acknowledged by the phone because it crashed, a denial of service attack is taking place on the phone. The authors question whether the telecom network will survive a large scale attack, where $10,000$ phones are reconnecting at the same time.

The attack described above directly targets mobile phones, especially via SMS on GSM networks. Presented attacks on the network itself is not as common and attacks targeting LTE are even rarer, probably because of it being too new and not yet widely implemented. The paper "Non-Access-Stratum Request Attack in E-UTRAN" [47] by Yu and Wen examines vulnerabilities in the LTE radio network E-UTRAN. It was found that IMSI numbers was transmitted in plain text without confidentiality and integrity protection, all according to the standard [26]. Collecting IMSI numbers in plaintext could be used to emulate several UEs and launch an attack from one device using multiple virtual UEs. With this technique, the MME assumes the messages are originating from multiple UEs and handles them accordingly. If an attack that requires multiple messages to be sent while the message could only be sent once per device, this technique could be used to send multiple messages from one device. Yo and Wen show that the network can be stressed heavily by sending attach requests for many virtual UEs with collected IMSI values. The network may become so overloaded that it cannot handle all messages, rendering the attack a denial of service attack.

3.2 Telecom Tools

To send arbitrary messages from a UE in a network such as EPS or GSM, where a UE is called Mobile Station (MS), necessary control over the radio stack of the hardware is needed. Typical UEs are devices specialized in mobile communication such as mobile phones and tables, but also hardware extensions to computers such as USB dongles for 3G or 4G internet connectivity. In most cases, the radio stack is not presented directly to the user facing system on the device to conform to the real time demands of the network and to isolate it from malicious uses. This separate device is called a baseband and is usually a separate chip called a baseband processor which communicates with the main operating system and transfers the interesting part of the radio traffic such as payload containing voice, text or IP packets.

Several attempts on reverse engineering the baseband exist to get control of the radio traffic sent and received. On top of this, several open source projects exists which have varying control of the radio stack. Most notably is an Open Source Mobile Communications (Osmocom) project, Osmocom Baseband (OsmocomBB) [11]. OsmocomBB is a GSM baseband implementation making it possible to run your own GSM stack on a supported baseband processor. The project has developed drivers for baseband processors on specific MSs and implemented the GSM protocol stack from the MS side. OsmocomBB provides custom firmware for the supported baseband

processors which provide a proxy of the GSM L1 interface towards a host computer running other OsmocomBB applications. One such application is the *mobile* application which implements large parts of a regular GSM MS [48]. This can easily be used to send and receive arbitrary SMS or MMS messages, establish voice calls and select cell to connect to. By implementing a new application, or extending an existing, any arbitrary layer two message can be sent via the L1 proxy. At the time of writing, OsmocomBB has varied support of at least 10 different MSs [49].

While OsmocomBB targets GSM, there exist tools that targets LTE. Since most open source development in this field relies on reverse engineering of equipment, research on LTE basebands is going forward as such equipment is getting more popular. The Osmocom team has shown interest in LTE, where one member has deployed his own LTE eNodeB [50]. P1 Security has been testing USB LTE devices and had the possibility to dump the baseband firmware as well as capturing LTE traffic from the device [51]. The OpenLTE project [52] aims to provide an open source implementation of the 3GPP LTE protocols. The focus is on transmission and reception of UE traffic and currently only supports handling and decoding LTE radio traffic. Compared to OsmocomBB, OpenLTE has much less functionality and cannot replace an LTE baseband yet.

3.3 Related Fuzzing Research

Robustness testing and fuzzing has a long history as described in Section 2.7.1. Some research has been made about integrating robustness testing in an existing testing framework, especially conformance test environments since that makes it easier for model-based fuzzing. The previous research presented here is using model-based fuzzing in TTCN-3 and shares a lot of goals of this thesis. They all have a goal of making use of the protocol models defined in TTCN-3 to construct messages with invalid data. However, they take different approaches of using the models and they differ in going for mutation or generation-based fuzzing.

In one paper [53], the authors present an approach in TTCN-3 to automatically extract the message structure from existing test data. The input syntax is then used to create invalid messages with mutation according to an attack heuristic algorithm. These predefined algorithms are type specific and cover common vulnerabilities such as buffer overflow patterns for the `charstring` TTCN-3 type. The authors state that the idea of this paper is to automate the process of fuzzing by using automatic construction of invalid messages, script generation for message injection via reuse of existing conformance test cases and verdict determination from the same conformance test. Since it is mutation-based, it is inherently dependent on valid test case data to create invalid messages.

A case study was presented in the paper where the concept was applied on the application layer control protocol Session Initiation Protocol (SIP) was used. Three SIP terminals were used as SUTs and all of them experienced some kind of unwanted behavior during the execution. They showed that the concept of using an automated approach to extract input syntax, generate malformed messages, inject them to the SUT and determine a verdict can be used for testing real world applications.

The DIAMONDS project [54] has a special focus on model-based fuzzing and presented [39] an extension to the TTCN-3 language for fuzz generation capabilities. The extension introduces a special fuzzing mechanism where a call to `fuzz()` could mutate or generate a value. The presentation shows that such a call could generate a new value without providing an existing value, if a suitable generator for this data type exists. The authors state that the generators and mutators should be implemented by making use of already existing fuzzing tools, such as Peach [55]. It is unclear if their extension could automatically generate used-defined models by e.g. traversing a `record` and generating its field using a suitable generator, or if the tester

has to define a generator for this special type. A goal of the extension is to make the use of fuzzing easy for existing TTCN-3 users. The DIAMONDS project has performed model-based fuzz testing at Ericsson of the eNodeB on protocol layers which were not covered by any existing tool [56]. The test found some faults and the authors concluded that it should be possible to test new protocols with reasonable effort.

3.4 Fuzzing Tools

There are several fuzzing tools available, both commercial and freer versions. To motivate the need of creating a new one, existing tools has to be researched to see if they match the requirements. Several tools presented here are model-based but often require the tester to write their own models, although some tools come bundled with models for a set of protocols. This means that most tools do not automatically support a new protocol. Some tools do support creating models via reverse engineering, by providing it with captured network traffic.

The Peach Fuzzer [55] is a general open-source fuzzing tool with both mutation and generation capabilities. Based on a user-defined model, it can generate messages both randomly and using some commonly vulnerable values. For each protocol to fuzz with Peach, a model has to be created. Creation of this can be a tedious act, since the effectiveness of the fuzzing will depend on the detail of the model. Radamsa [57] is another general fuzzing tool, specialized in mutation-based fuzzing. According to the authors, it is simple to setup and use. It contains several mutation algorithms and it can be easily scripted. However, it only works on given sample inputs, and it does not have any specific telecom support. Another well-known open source tool is Sulley [58] which monitors network traffic and automatically modifies the traffic by mutating and changing order of messages. The models of a protocol have to be provided in Python code, which defines the fields of a message and whether it should be fuzzed or not.

Commercial model-based fuzzing tools also exist, which have some differences from the open source tools. The most common difference is that they come bundled with many more protocol models to perform fuzzing on. Codenomicon Defensics [5] is such a commercial model-based fuzzing tool which contains over 200 protocol test suites of which 13 are telecom specific, such as GTPv1 and Diameter [59]. However, there is currently no support for LTE application layer protocols, such as S1AP or NAS. Codenomicon offers the Defensics Traffic Capture Fuzzer [60] which performs fuzzing on protocol without preexisting models. It works by analyzing traffic captures of said protocol to create an approximate model. This creates a model much quicker than manual creation, but carries the same disadvantages as other mutation-based fuzzers. P1 Telecom Fuzzer [6] is another commercial fuzzing tool specifically targeting telecom networks. 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. Both Codenomicon Defensics and P1 Telecom Fuzzer provide protocol test suites, but neither provides functionality to adapt fuzzing to a new protocol based on its specification. While Codenomicon offers fuzzing based on traffic capture as described earlier, P1 Security state that a new protocol is modeled manually in one to five days upon request.

Chapter 4

Design and Implementation

The design of the fuzzer consists of two parts: investigating and analyzing the existing test environment, and dividing the fuzzer into components and deciding requirements of the components based on the existing test environment. The first part is described in Section 4.1 where the two existing test environments of the Ericsson SGSN-MME are described: TTCN-3 and GTT. The second part is described in Section 4.2 where the fuzzer is divided into several components with requirements to get it fully integrated with the environment.

4.1 Existing Test Environment

There exist two test environments at Ericsson for the SGSN-MME: TTCN-3 and GTT. Both TTCN-3 and GTT is used for functional testing of the MME, but they have some differences. A brief explanation on how they both are used is presented below with an evaluation on what would fit the fuzzer best.

4.1.1 TTCN-3 Environment

TTCN-3 is used for black-box conformance testing and, primarily, only looks at messages that go in and out from the test case to the SUT. The environment is based upon TITAN, described in Section 2.3.3, and has functionality to simulate a complete EPC network where the different nodes surrounding the MME (such as the SGW and HSS) can be implemented in TTCN-3. The environment makes sure that the interfaces of the MME are mapped so that messages are routed from the testing environment to the SUT and vice versa, to create a complete and isolated EPC network.

Two different TTCN-3 testing frameworks exists for testing the SGSN-MME: FAST and Inka. The FAST framework has a lot of functionality separated into different functions, which combined can make a complete traffic flow test case, e.g. 1) attach a UE, 2) update tracking area, 3) transfer to other MME, 4) detach. This makes it easy to write new test cases by making use of the functionality that already exists, as well as adding the small parts not implemented. Since it has a lot of functionality, it is quite large in size and could be hard to understand. Inka on the other hand is a new framework, a lot smaller and without a lot of functionality. The focus is instead to give the tester full control over the signaling, which leads to the requirements of handling every message going in and out.

4.1.2 GTT Environment

GTT is an Ericsson internal test framework specifically for the SGSN-MME. It has many similarities with TTCN-3 described in Section 4.1.1; it performs testing by sending messages and verifying the received messages, as well as simulating the nodes around the SGSN-MME to build a complete EPC network. However, GTT also performs white-box testing by reading various variables, counters and gauges from the SUT to determine if the SUT is in an expected state.

The test cases are written in Erlang and it is fairly easy to write simple test cases. GTT supports testing of all interfaces of the SGSN-MME. Due to its close connection to the SUT, some messages are not generated and sent over the real interfaces as e.g. SCTP packets over IP, but instead injected into the system and passing over some of the input message parsers in the SUT. Therefore, it might be hard to generate messages that test the parsers. Though for some interfaces and protocols the messages are sent over real links with separate encoders and decoders for these messages, such as for the S1-MME interface which is described in Section 2.2. At the time of evaluating the testing environments, it was hard to run GTT tests on a real node and it was necessary to use simulated hardware. This might introduce false-positive test results due to the simulated hardware not performing as real hardware would; for example the network and routing might be different leading to messages taking different paths.

4.1.3 Evaluation

The environments were evaluated to make a decision whether to integrate fuzzing in TTCN-3 or GTT. The key advantages and disadvantages found are summarized and presented below for both TTCN-3 and GTT.

TTCN-3 advantages

- Full protocol specification models
- Negative testing functionality available in TITAN
- Traffic flow functionality available in FAST
- Fuzzing with TTCN-3 is being researched, even inside Ericsson
- External C++ code can be integrated with TITAN generated C++ code

GTT advantages

- Easy to use
- Supports almost all interfaces
- Separate encoder and decoder exists for S1-MME interface
- Any Erlang code can be integrated with test suite

TTCN-3 disadvantages

- Negative testing in TITAN is not fully explored at Ericsson
- Need to deal with many signals with Inka, no traffic flow and not in parallel mode
- FAST might be hard to understand due to its large size

GTT disadvantages

- Shared memory for communication
- Need encoder and decoder for protocols to generate messages
- May return false-positives due to simulation
- Difficult to run with real hardware

A unanimous decision was made together with experts in the respective testing framework. TTCN-3, specifically with TITAN, turned out to be the preferred testing language for model-based fuzzing due to having full protocol specifications that already are modeled in the language. The simplicity to create external C++ code to integrate with the generated TITAN code and the fact that fuzzing is being researched inside Ericsson also made a big impact in the choice of testing framework. Too much work would have to be done outside the scope of this thesis to be able to use fuzzing within GTT. This was the main argument for the decision to choose TTCN-3 as the testing framework to implement the fuzzer in. In order to reuse as much functionality as possible about the protocols, the FAST framework was further selected. It was seen as more supported and while it is of large size, all parts do not need to be investigated nor used. Inka seemed too complicated with the signal handling, which would make fuzzing more difficult.

4.2 Design Requirements

The aim was to integrate a model-based fuzzer with an existing testing environment, which after the evaluation presented in the previous section led to the choice of integrating with TTCN-3 and TITAN. While there was some negative testing functionality in TITAN, it was limited to override fields with constant values. These constant values had to be provided at compile-time, which was seen as an impossible task for a fuzzer due to the amount of combinations that a fuzzer would generate. This also led to the need of being provided a message in which some fields could be overridden, which would limit the amount of fuzzed messages to the amount of provided messages.

A brief introduction to the TTCN-3 test environment used at Ericsson for conformance testing and the requirements that has to be fulfilled before fuzzing can be applied will be described in the following subsections.

4.2.1 Conformance Test Environment

The existing environment for functional and conformance testing can be seen in Figure 4.1, which illustrates the main components of the test environment. TITAN takes as input a test suite consisting of a set of TTCN-3 modules with models and test cases, and ASN.1 modules [34]. It then parses and analyses the input modules and generates C++ code. The C++ code for the modules are then compiled to an ETS together with external C++ code such as test ports and the base library. Test ports used by the ETS are defined in C++ and are a realization of TTCN-3 test ports [33]. The test port creates the bridge between the ETS and the SUT for external communication. The base library consists of C++ classes which defines the basic TTCN-3 data types and built-in functions such as timers, ports, test components and verdict handling. Also included in the base library are necessary functions to run the ETS which comprises configuration routines and logging procedures.

The generated ETS contains all necessary functionality for setting up the execution of the test. This includes encoding and decoding of messages, logging and SUT communication. During execution, messages will be sent to the SUT which will respond accordingly. Each test case will be handled separately and a restart of the SUT will be established if a test case caused any error to not affect the next test case. At the end of the execution run a summary is presented of the executed test cases. At this point, appropriate actions can be taken by the tester according to the test case verdict (pass or fail).

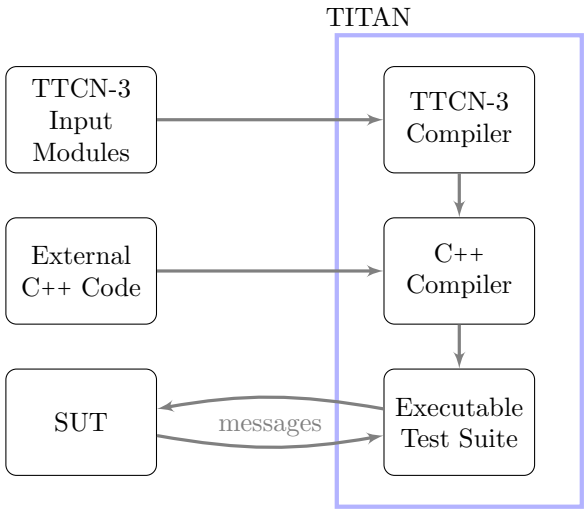


Figure 4.1: Conformance test environment

4.2.2 Fuzzing Test Environment

To integrate fuzzing in the conformance test environment, the environment needs to be extended with new functionality. The proposed extension components will be described below, and an illustration of the extended environment can be seen in Figure 4.2.

The existing functionality for negative testing only allowed constant values provided at compile-time. A test case would have to be set up for each fuzzed message which would be time consuming and not very efficient. Instead the fuzzed messages have to be dynamically generated at run-time. To generate messages at run-time, the TTCN-3 models have to be extracted at compile-time to have necessary information available at run-time. The models will be used as input to a fuzzing engine which has the functionality to generate and populate the extracted models.

At last, an observer has to be designed to distinguish if the SUT could handle the malformed message correctly. The observer will monitor specified properties of the SUT and will make a decision based on these if the fuzzed message caused any problems. The observer will give feedback to the ETS which can react accordingly, e.g. save the message that caused the problem together with useful information such as crash logs.

4.2.3 Model Extraction

To use model-based fuzzing, the models of the protocol need to be extracted. When testing protocols in TTCN-3, the protocol are modeled in detail using TTCN-3 data types and structures. As described earlier, the fuzzer needs to have the models at compile-time to have all the required information available at run-time.

The model extractor component has the responsibility of extracting the models from the TTCN-3 files to a suitable format for the fuzzing engine. To get good test coverage, all types and structures in a model need to be extracted. In order to create a generation-based fuzzer, the extractor should not be dependent on existing populated models as mutation-based fuzzers are. Instead, all necessary information should exist to build a message from scratch. To summarize,

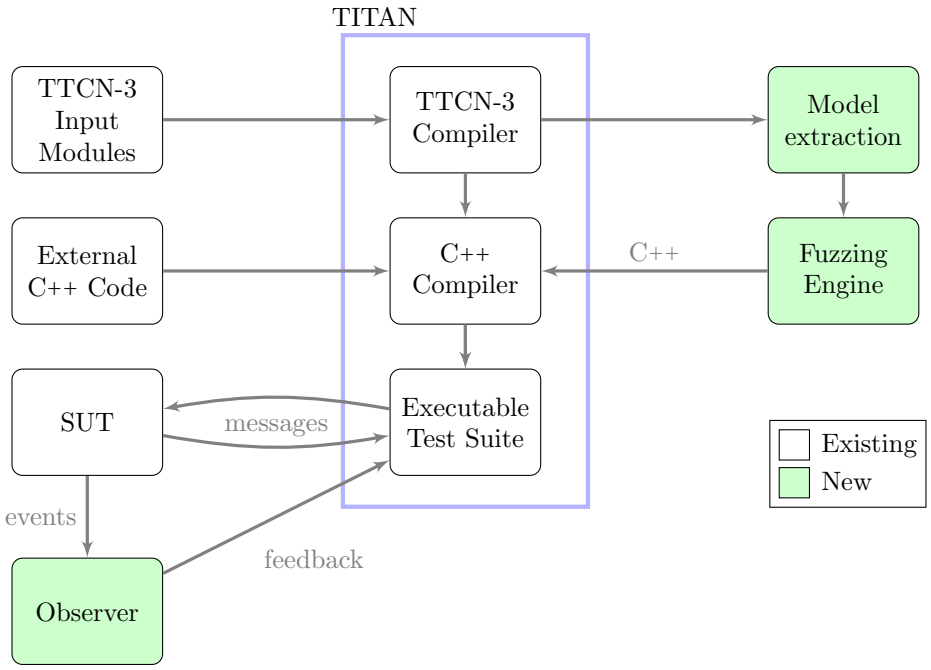


Figure 4.2: Extended test environment with fuzzing components

the requirements for the model extractor were the following:

1. extract all subtypes defined,
2. extract all structure types like: `record` and `union`,
3. extract optionality of fields in structures, and
4. independent of existing populated models.

4.2.4 Fuzzing Engine

To make use of the models in fuzzing, the models need to be populated with data to create an instance of the model. It is assumed that the model extractor provides the fuzzing engine with the models in a suitable format containing all necessary information. The fuzzing engine then needs to have the ability to create instances of all TTCN-3 types, both built-in and extracted subtypes. The requirement for this generation is for simplicity only set to generate random values. The fuzzing engine should also be able to create instances of structured types, filling the fields with generated values of correct types.

Another set of requirements of the fuzzing engine covers the ability to control the randomness in some way. One such requirement is being able to easily reproduce generated messages. To do this, the random generation needs to depend on a seed than can be set before generation where the same seed will lead to the same generation. Another required feature is the ability to preserve certain field from randomization, such as fields that the tester for some reason wants to be statically set. One typical example of this is fields containing some authentication information, where invalid data may result in the message being thrown away by the SUT due to authentication failure.

```
1  module Example
2  {
3      type bitstring BIT4 length(4);
4      type hexstring HEX2 length(2);
5      type record MSG
6      {
7          BIT4    fourBitField,
8          HEX2    twoHex,
9          integer number optional
10     };
11 }
```

Figure 4.3: Example TTCN-3 module with type definitions

In order to get an integration with the TTCN-3 environment, the fuzzer needs to be used from within TTCN-3. To do this, the functionality of the fuzzing engine needs to be provided to the TTCN-3 environment and not be a standalone tool.

To summarize, the requirements for the fuzzing engine were to:

1. generate random value for any built-in TTCN-3 type: `integer`, `bitstring`, ...,
2. generate random value for any extracted type, including structures,
3. preserve certain values from randomness, such as authentication fields,
4. have reproducible results by setting seed, and
5. be usable from inside TTCN-3.

4.2.5 Observer

While running functional black-box tests, the verdict of the test is often determined by comparing the messages received from the SUT to some expected value. In negative testing such as fuzz testing, no particular response is expected. Instead, some kind of error condition within the SUT is researched. A black-box approach would be to validate if proper actions are still possible to perform, but that will only determine if the SUT is still available. To determine if some internal fault has occurred that is not visible from the outside, a white-box approach is needed that uses information from within the SUT. The observer is an white-box approach which monitors the system while performing the fuzz testing to figure out if there has been some error.

However, the observer is dependent on the SUT, which means that it is impossible to create a generic monitor which covers all SUTs. Therefore, the monitor part was not designed nor implemented generically but only with the requirement that it should provide useful feedback from the SUT to the fuzzing test case.

4.3 Implementation

To fulfill the requirements described in Section 4.2, some implementation choices were made. One of the delimitations of the thesis (see Section 1.2) was to either integrate an existing fuzzing engine or to develop one from scratch. With the requirements fully stated in the design section,

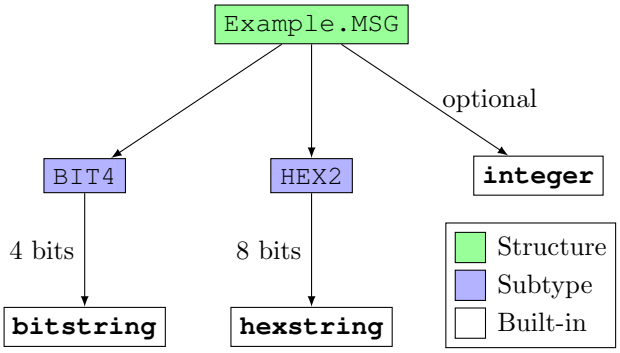


Figure 4.4: Extracted model tree of MSG structure in Example module

several open source fuzzing engines, e.g. Peach [55], fulfilled most of the requirements. However, none of them had the possibility to integrate with TTCN-3 without heavy rewriting since they were constructed as standalone tools. Also due to licensing issues, integration was not possible for all fuzzing engines. Therefore, a simple fuzzing engine was developed entirely for the fuzzer, which may lack features that existing fuzzing engines have.

Since the choice of TITAN was made, the fuzzer's components were implemented in C++ out of convenience. TITAN allows external C++ code to be used when compiling the ETS, which makes it easy to integrate the fuzzer with the compiler. Since the goal was to integrate the fuzzer with the existing environment, an Application Programming Interface (API) to use the fuzzer from TTCN-3 was developed to create a complete integration. The implementation of the components described in Section 4.2 and the fuzzing API are described below.

4.3.1 Model Extraction

The TTCN-3 files containing message structures has to be parsed for the models to be extracted. Since TITAN already parses the files to generate C++ code it is unnecessary to create a separate TTCN-3 parser. Instead, the generated C++ code was statically analyzed at compile time to extract the models. The TTCN-3 models consist of structure types and subtypes.

As described in Section 2.3, there are several built-in types in TTCN-3 and users can define subtypes of these with restrictions on its length or value. For example, a subtype of `bitstring` may have a length restriction of exactly four bits, or a subtype of `integer` may have a value restriction of positive values up to 255.

An example of a TTCN-3 model is illustrated in Figure 4.3. Two subtypes are defined in the example, both with a length restriction. The first is a `bitstring` named `BIT4` with a length restriction of four bits. The second subtype is a `hexstring` named `HEX2` with a length restriction of two hexadecimal digits, which is eight bits. A `record`, which defines the message structure of this illustration, named `MSG` can be seen on line 5. The user defined subtypes `BIT4` and `HEX2` together with an optional field, of type `integer`, form the field types of `MSG`.

When extracting the message structure, the model extractor builds a tree with a record structure as the root. A tree of the example module presented earlier is illustrated in Figure 4.4 with the message `MSG` of module `Example` as the root. The `Example.MSG` has a weighted branch to each type defined in `MSG` which states if the field is optional or not. If a node is a subtype it will have a branch to a leaf with a length restriction of the specified built-in type. The tree structure can also have nested types with a `record` within another `record` but the leaves will always be a built-in TTCN-3 type.

4.3.2 Fuzzing Engine

The generation of data takes place in the fuzzing engine where generators for the extracted subtypes and structures are created at compile-time from the extracted trees. Together with generators for the built-in TTCN-3 types, this is returned back as C++ code to the TITAN compiler which includes it in the ETS.

Generation of structures is made by traversing the model tree and generating values for each of the leaf nodes. This way a completely generated structure with all fields filled will be produced. If the branch marks the field as optional, a random boolean value will decide if the field is generated or omitted. All structure fields have names, and the tester may choose to override certain fields to preserve them from randomization. When generating a field in a structure, the name combined with the type of the field is looked up in an override table. If a value exists in the table, this value is used instead of generating a new one. This can be useful to preserve values from testing, such as identification or authentication fields. The `union` and `enum` types will be generated by choosing one possible alternative at random.

The generators for the subtypes also traverse the extracted tree, with the node pointing to a built-in type to be generated. The generator for this type is called with any requirements stated in the branch to the leaf. The result of this generation is returned as the result of the subtype generation.

The built-in types are generated by creating a number of random bits. All built-in type generators require a size parameter, indicating how many bits to generate. If a `charstring` with 12 characters is to be generated, the generator produces 96 random bits which are used to create a `charstring`. The same procedure applies for the other built-in types, such as generating 32 random bits for a 32-bit `integer`. The random bits returned are based on a random seed that can be set at any time. This way, the same results can be reproduced at any time given the same seed.

4.3.3 Fuzzing API

To use the fuzzer, glue code between the TTCN-3 code and the C++ code is generated. This glue code exports the generator functions from C++ to TTCN-3. To generate the glue code, a generator is called with the modules that the fuzzer should handle as input. The glue code contains functions that are common to the fuzzer and the specific input modules. The glue code consists of two parts, one generated TTCN-3 file and one generated C++ file.

Using the module `Example` in Figure 4.3 as input for the fuzzing glue generator, the TTCN-3 module with exported functions seen in Figure 4.5 will be produced. The functions on lines 9 to 15 provide the functionality of overriding certain values in a record. A function call to `fuzz_override_INTEGER("number", 4)` will make sure that later generated structures that contain a field named *number* have the value 4. This also applies to subtypes of the native types, so any subtype of `integer` called *number* would have the value 4.

The functions on lines 17 to 23 generate new values with the type according to its name. All functions takes an `integer` value *bits* ≥ 0 that determine how many bits that should be generated, except `fuzz_gen_BOOLEAN()` which is always a single bit. A function call to `fuzz_gen_INTEGER(n)` would generate an `integer` m, $-2^{n-1} \leq m \leq 2^{n-1}$. Internally in the fuzzer, the least significant n bits are extracted from the generated bits. The n:th bit is interpreted as a signed bit, where 0 is positive and 1 is negative. Calling `fuzz_gen_BITSTRING(4)` would generate a random string of four bits, e.g. `'0110'B` expressed in TTCN-3. To get a random number of bits, the argument *bits* could be the return value of `fuzz_gen_INTEGER`. For example, `fuzz_gen_BITSTRING(fuzz_gen_INTEGER(4) + 8)` would produce a string of n bits, $1 \leq n \leq 15$.

```
1  module fuzzGlue
2  {
3      import from Example all;
4      // Built-in functions
5      external function fuzz_init();
6      external function fuzz_end();
7      external function fuzz_set_seed(integer seed);
8      // Overrides for native types
9      external function fuzz_override_BOOLEAN
           (charstring name, boolean val);
10     external function fuzz_override_INTEGER
           (charstring name, integer val);
11     external function fuzz_override_FLOAT
           (charstring name, float val);
12     external function fuzz_override_CHARSTRING
           (charstring name, charstring val);
13     external function fuzz_override_BITSTRING
           (charstring name, bitstring val);
14     external function fuzz_override_HEXSTRING
           (charstring name, hexstring val);
15     external function fuzz_override_OCTETSTRING
           (charstring name, octetstring val);
16     // Generators for native types
17     external function fuzz_gen_BOOLEAN()
           return boolean;
18     external function fuzz_gen_INTEGER(integer bits)
           return integer;
19     external function fuzz_gen_FLOAT(integer bits)
           return float;
20     external function fuzz_gen_CHARSTRING(integer bits)
           return charstring;
21     external function fuzz_gen_BITSTRING(integer bits)
           return bitstring;
22     external function fuzz_gen_HEXSTRING(integer bits)
           return hexstring;
23     external function fuzz_gen_OCTETSTRING(integer bits)
           return octetstring;
24     // Generators for Example
25     external function fuzz_gen_Example_BIT4() return Example.BIT4;
26     external function fuzz_gen_Example_HEX1() return Example.HEX1;
27     external function fuzz_gen_Example_MSG() return Example.MSG;
28 }
```

Figure 4.5: TTCN-3 glue code for Example module

The functions on lines 25 to 27 are specific for the `Example` module. Note that these functions do not take any arguments such as number of bits, because the types are defined with length specifications. Calling `fuzz_gen_Example_BIT4()` would generate a correct `BIT4` value which is a bitstring with length of four bits. Calling `fuzz_gen_Example_MSG()` would generate a `MSG` record value. All fields inside the record will be generated; a generated `BIT4` for the `fourBitField` field, a generated `HEX2` for the `twoHex` field and a generated `integer` for the `number` field. Since the `number` field is optional, a random boolean value will decide if it is filled with an `integer` value or `omit`. Since the `number` field also is of type `integer`, and no subtype with length restriction, a default bit length of 32 bits will be used by the generator.

```
1  module ExampleTesting
2  {
3      from Example import all;
4      from fuzzGlue import all;
5      // Port definition
6      type port ExamplePort message
7      {
8          inout MSG
9      };
10     testcase fuzzing()
11     {
12         fuzzInit();
13         port ExamplePort sutPort;
14         // ... connect sutPort ...
15         var MSG fuzzed_message := fuzz_gen_Example_MSG();
16         sutPort.send(fuzzed_message);
17         // ... evaluate ...
18     }
19 }
```

Figure 4.6: Example TTCN-3 test case using fuzzing functions

An example fuzzing test case that uses the exported generators may look like the one in Figure 4.6. The module `ExampleTesting` has a port to an SUT that communicates with the `MSG` type as the protocol. In the test case `fuzzing`, the fuzzing engine is initialized on line 12 which is required prior to any other fuzzing function call. A `MSG` message is then randomly generated by the fuzzing engine on line 15 and sent on line 16 to the SUT. After the message has been sent, the tester may for example evaluate responses from the SUT or use an observer to get information about how the SUT reacted to this random message.

Chapter 5

Case Study: NAS on Ericsson MME

The fuzzer was tested on the NAS protocol [26], described further in Section 2.2. The experiment on NAS was performed in several steps along the development of the fuzzer. We selected NAS as a case study since previous research has shown that most attacks originate from end-user equipment, as presented previously in Section 3.1. NAS directly connects the UE with the MME, and the MME is a critical network element in EPS. Through this connection, the MME may accept potentially untrusted input directly from UEs. In addition, NAS is a relatively simple protocol that has a fairly low number of message types and the majority of the protocol fields consist of bit strings. This means that fuzz testing can easily get good coverage due to the low number of different types to test. NAS is also already used in current conformance testing at Ericsson. Thus, the protocol is already modeled in detail in TTCN-3 and a test framework for functional testing exists.

The development of the case study was divided into three phases. The goal of the first phase was to make sure that the fuzzer could generate NAS messages and that these could be sent. This was tested during development of the fuzzer to make sure that the fuzzer could handle the NAS TTCN-3 models. When the fuzzer got stable enough to generate NAS messages properly, a test case to send generated messages and receive answers was developed. The second phase extended the test case with the observer component to actually find possible errors occurring. When the test case could find errors and return the messages causing errors, the third and last phase was entered. This phase evaluated the possibility of using the fuzzer to create a larger impact than the errors that was found in the second phase, such as creating a denial of service attack.

5.1 Test Setup

The Ericsson SGSN-MME is heavily tested with TTCN-3, and this is the case with the NAS protocol stack. NAS is modeled in TTCN-3 and covers every message type and field that is valid in NAS, according to its standard [26]. In practice, this is done by having a base NAS structure called `PDU_NAS_EPS`. This structure contains the header common for both ESM and EMM as described in Section 2.2.1 with a *Protocol Discriminator* field. The other field is a `union` which states if the message is an ESM or EMM message. An EMM message will contain a *Security Header Type* and an ESM message will contain an *EPS Bearer Identity*. Both EMM and ESM

have their own union field which states what message type the data consists of. The actual data is defined in TTCN-3 as their own `record`.

When the fuzzer is provided the NAS TTCN-3 module as input, the models are extracted as described in Section 4.3.1. Since all message types and fields in the NAS module are defined with TTCN-3 types, all possible variations of NAS messages are possible to generate with the fuzzer. The glue code generated, as described in Section 4.3.3, will contain functions to generate a specific message as well as a function to generate a complete `PDU_NAS_EPS` message with a randomly chosen message type.

To test the Ericsson SGSN-MME NAS stack, generated messages has to be sent to the SUT. The conformance test framework FAST, as described in Section 4.1, provides functionality to send and receive messages as a UE and an eNodeB. This framework is used to simulate the needed components around the MME, such as the SGW and HSS, as well as to simulate a UE and an eNodeB. Generated NAS messages can then be sent with the sending functionality of FAST.

5.1.1 State Awareness

To get better results, a certain state of the UE was wanted before sending messages. This method would more likely be able to find system implementation faults than simply sending messages without any prerequisites. Both the EMM and ESM protocol are built uppon state machines. The UE and the MME both have one EMM state machine, and one ESM state machine per bearer context. The state machines of the protocols are described further in Section 2.2. To get as many messages accepted as possible, the first step is to get the UE recognized by the MME by attaching it. Therefore, this experiment is focusing on the EMM state machine, since this machine is responsible for the attachment and detachment of the UE.

A simplified diagram over the EMM state machine for the MME can be seen in Figure 2.7. The main states are *EMM-REGISTERED* and *EMM-DEREGISTERED*, which describes if the UE is attached or detached respectively. The UE can also be in either *EMM-IDLE* or *EMM-CONNECTED* mode, which represents if there is an NAS signaling connection between the UE and the MME, regardless of the EMM state. According to the NAS standard [26], when the UE is in *EMM-IDLE* mode it is only allowed to send initial NAS messages that can be used to go from *EMM-IDLE* to *EMM-CONNECTED* mode. In *EMM-CONNECTED* mode however, several more messages are allowed to be sent and received by the UE, depending on the EMM state. The most common signaling is made in *EMM-REGISTERED*, which would mean that the combination of *EMM-REGISTERED* state with *EMM-CONNECTED* mode is the best choice for sending fuzzed messages.

To get our simulated UE attached in the *EMM-REGISTERED* state, the attach sequence is performed as is described in Section 2.2.4. With the use of the existing test framework FAST, the attach sequence is already implemented and can be reused easily by simply calling a function. This function will perform the attach sequence and put the UE in *EMM-REGISTERED* state and *EMM-IDLE* mode. To get to *EMM-CONNECTED* mode, a Service Request is sent and upon arrival of a Service Accept message the transition to *EMM-CONNECTED* mode is made. The functionality of the FAST framework to send messages is used both for sending the Service Request and the fuzzed messages. To know what EMM state and mode the MME believes the UE to be in after sending the message, the responses has to be handled. A receive window of 200 milliseconds after sending handles some specific messages, such as a Detach Request to the UE or an S1AP Context Release sent to the eNodeB. By handling some of these specific messages, enough is known to conclude if the UE is still in the wanted *EMM-REGISTERED* state and *EMM-CONNECTED* mode.

5.1.2 Encryption

During the attach procedure, certain security functions are also performed. This includes setting up ciphering keys between the UE and the MME. These keys are later used to encrypt messages transferred between the UE and the MME. Both the MME and the UE will, according to the NAS standard [26], dismiss all encrypted messages that cannot be decrypted without errors. Errors include using other keys than negotiated and invalid message authentication codes. As soon as ciphering keys are set up, NAS messages are supposed to be encrypted and rejected if they are not.

For the fuzzer, encryption becomes as important as entering states. The purpose of entering a certain state before sending generated messages was to get as many messages handled by the SUT as possible. Since the standard states that only encrypted messages are allowed after setting up ciphering keys, the fuzzer need to be aware of these to encrypt the generated messages. Without encryption, the SUT will only be tested of its ability to discard unencrypted messages. With encryption, the SUT will decrypt the generated message properly and handle it. By using the FAST framework for sending messages, functionality for handling ciphering keys, encryption and decryption of messages is provided in the sending functionality.

5.2 Observer Implementation

As described in Section 4.2.5 an observer is needed to monitor the SUT for faults. To construct an observer, implementation specific information about the SUT is required to get useful feedback. The software design of the SUT is described in Section 2.5.2, and based on this it was deemed to be very interesting to look for Erlang worker crashes. To monitor Erlang crashes on the SUT, the observer needs to get enough information to conclude that a crash has occurred. The first try of implementing this was to monitor the crash log files, but it turned out that this log file was not written to instantaneously but delayed and crash reports were grouped together if they were similar. Instead, the second try consisted of checking a crash counter to get the total number of crashes that has occurred. The crash counter is an integer value that increases for every Erlang crash, regardless of the cause or impact of the crash. By polling the crash counter on demand, the observer could conclude if and how many crashes has occurred since last check. Since the environment is completely isolated, it is only the fuzzing test that can affect the SUT and the crash counter.

When implementing the observer, it was found that polling the crash counter was quite time consuming. The test case sends messages every 200 milliseconds, and a crash counter poll take approximately 1 second. If the crash counter is checked after every message, the throughput goes down to 50 messages per minute from 300 messages per minute without the check. To increase the throughput while checking the counter, a bulk of messages is sent and the crash counter is checked before and after sending. If the crash counter has increased, a crash has occurred and a binary search on the sent messages will find the message that caused the crash. To divide the list in even parts for binary search, the message list has to be of length 2^X. By using $X = 4$, the counter is checked after 16 messages and the throughput is instead approximately 228 messages per minute.

The test case with an observer is described in pseudo-code in Figure 5.1. The test case consists of an infinite loop, using generated NAS messages from the fuzzer on line 4, sends them using a helper function on line 5, stores them in a list on line 6, and finds the messages causing crashes on lines 7 - 13 as soon as the number of messages sent are 2^X. The function `SendAndHandle`, on lines 14 - 29, takes care of being in a wanted EMM state and mode before sending a message. It also handles incoming messages after sending m, to conclude what EMM state and mode the

UE is in. The wanted EMM state *EMM-REGISTERED* is reached by trying to attach if the UE is in any other state, see line 16. The wanted EMM mode *EMM-CONNECTED* is reached by sending a *Service Request* when in the correct state, see line 19. When this criterion is met, the message is sent on line 28 and incoming messages are handled by calling `HandleIncoming` on line 29.

The procedure for finding sequences of messages causing crashes is described in pseudo-code in Figure 5.2. The reproduction function takes as argument the expected number of crashes, W, which initially is the increase of the crash counter value, and the list of messages M. It starts by splitting the list of messages into two lists, M_1 and M_2, and sends all the messages in them using the helper function described earlier, counting the number of crashes created by each of the lists, see lines 6 - 14. If a crash was produced by the first part, this list is used as input to the function recursively with the measured expected number of crashes, see lines 16 - 19. The same procedure happens for the second part as well, see lines 20 - 23. The result from the both recursive calls is stored in the list S which will be a list of smallest possible sequences of messages causing a crash. If the two lists from the splits together were unable to create the expected number of crashes, this implies that some crashes were not reproduced due to splitting the list. If this happens, the input list is also added to the list S on lines 24 - 26. The list S is in the end returned on line 27.

5.3 Large Scale Attack

To further investigate the robustness of the SUT, test cases that make use of the information gathered by fuzzing to create a large-scale attack were created. The idea of creating a large-scale attack was to use the fuzzing test case described in Section 5.2 to get find a message which will cause a process crash, and exploit the worker crash escalation policies. By creating multiple worker crashes within a certain time slot, not only the workers will restart but the supervisors as well. Going further, when enough supervisors restart at one level, their supervisors will restart, and so on. The aim of the attack is to get as high up in the supervisor tree as possible, hopefully high enough to disconnect all attached UEs rendering the attack a full scale denial of service attack.

The workers are identified in Section 2.5.2 as dynamic worker processes representing an attached UE. Therefore, to create several worker crashes in a time period, several UEs have to send the message which caused a crash. While this could be done by a single UE repeating the process of attaching and sending the message, the attach phase takes too much time to complete to be able to send enough messages. This leads to the requirement of having multiple UEs attached where each of them sends the message.

One problem is to find out how many UEs that are needed to cause a crash. The solution depends on two things:

1. how fast the complete attack process takes for one UE, and
2. how many crashes that are needed to trigger an escalation.

The first factor is solved by running the test and measuring how long time an attack process takes when having different numbers of UEs attached. To solve the second factor we have to investigate the escalation policy of the system, which could be almost impossible without access to the system, its internal documentation and the source code.

If a message which could provoke a static worker crash was found, the escalation procedure would be rather simple to trigger due to the direct access to a supervisor restart. However, the generated messages, which have tested the system implementation so far, were not able to create a static worker crash and the large scale attack would have to be performed with an escalation

External: `send(`n`)`: FAST send function, sends NAS message n, encrypted if needed
External: `receive()`: returns the received NAS message n
External: `GenerateNAS()`: returns a generated NAS message n
External: `NASType(`n`)`: returns the NAS message type t
External: `crashCounter()`: returns an integer number of crashes
External: `AttachProcedure()`: FAST attach sequence, modifies global EMM state E_s and EMM mode E_m
External: `HandleIncoming(`t`)`: handles incoming messages for t milliseconds, modifies global EMM state E_s and EMM mode E_m
Global: E_s: current EMM state, initially EMM-DEREGISTERED
Global: E_m: current EMM mode, initially EMM-IDLE
Local: M: list of messages sent, initially $\emptyset$
Local: F: list of found faulty messages, initially $\emptyset$
Local: C: saved crash counter value
Local: X: denoting the 2^X number of messages sent before searching for errors, set to 4
Local: Z: denoting the time to wait for receving messages, in milliseconds, set to 200

```
1  Function TestCaseObserver() begin
2      C ← crashCounter()
3      while true do
4          g ← GenerateNAS()
5          SendAndHandle(g)
6          M := M ∪ {g}
7          if |M| = 2^X then
8              C' ← crashCounter()
9              if C' > C then
10                 f ← FindError((C' - C), M)
11                 F := F ∪ {f}
12             C := C'
13             M := ∅
14 Function SendAndHandle(m) begin
15     if E_s ≠ EMM-REGISTERED then
16         AttachProcedure() // Hopefully sets state EMM-REGISTERED
17         return SendAndHandle(m) // Recursively wait for correct state
18     else if E_s = EMM-REGISTERED and E_m ≠ EMM-CONNECTED then
19         send(ServiceRequest)
20         n_i ← receive()
21         if NASType(n_i) = ServiceAccept then
22             E_m := EMM-CONNECTED
23         else
24             E_m := EMM-IDLE
25         return SendAndHandle(m) // Recursively wait for correct state
26     else
27         // E_s = EMM-REGISTERED and E_m = EMM-CONNECTED
28         send(m)
29         HandleIncoming(Z)
```

Figure 5.1: Sequence of the test case with observer

```
External: crashCounter(): returns an integer number of crashes
External: split(M): returns ⟨M1, M2⟩ with the first half M1 and the second half M2
          of the list M
External: append(F,G): returns the combined list of F and G
Local: M: a list of 2^n messages
Local: W: expected number of crashes
1  Function Reproduce(W,M) return S // list of sequences of messages
2  begin
3      if |M| < 1 then
4          return ∅
5      C ← crashCounter()
6      ⟨M1, M2⟩ ← split(M)
7      foreach m in M1 do
8          SendAndHandle(m)
9      C' ← crashCounter()
10     C1 := (C' − C) // crashes from M1
11     foreach m in M2 do
12         SendAndHandle(m)
13     C'' ← crashCounter()
14     C2 := (C'' − C') // crashes from M2
15     S := ∅
16     if C1 > 0 then
17         // Find crashing sequences in M1 and add to S
18         S1 ← Reproduce(C1, M1)
19         S ←append(S,S1)
20     if C2 > 0 then
21         // Find crashing sequences in M1 and add to S
22         S2 ← Reproduce(C2, M2)
23         S ←append(S,S2)
24     if (C1 + C2) < W then
25         // Add this sequence if crashes after splitting are fewer
           than expected
26         S ←append(S,M)
27     return S
```

Figure 5.2: Procedure to find the smallest possible sequences of messages that cause a crash

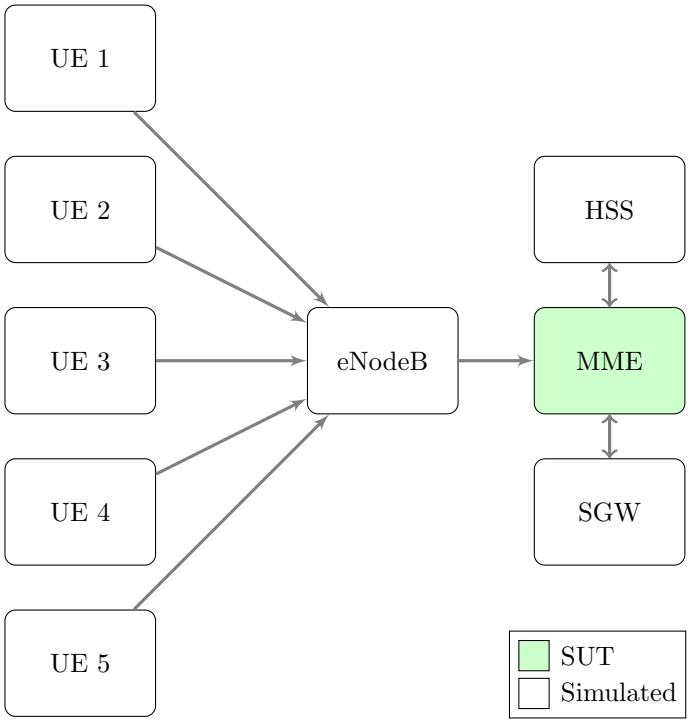


Figure 5.3: Large scale attack illustration

of dynamic worker crashes. To accomplish this, a certain number of dynamic crashes have to be provoked during a strict time interval. This leads to the requirement of attaching multiple UEs to reach the desired crash frequency. In the test setup, multiple UEs will attach to the MME via one eNodeB as illustrated in Figure 5.3. Each attached UE will send the same message sequence which the fuzzer has identified as a malicious sequence. The malicious message sequence will cause a dynamic worker to crash. To further enhance the crash frequency, a predefined number of UEs was attached before the attack began. In this way, waves of malicious messages were sent to the SUT. Each wave contained the same number of attached UEs, which after a couple of waves reached the desired number of crashes to trigger a dynamic crash escalation. When the UE have sent the malicious message it will be detached because the dynamic process which handled the connection crashed. The UE have to reattach in order to participate in a new attack wave. The time it took for a UE to attach and send the messages corresponds to the crash frequency of this test. This frequency has to exceed the configured crash frequency of the SUT to reach a dynamic worker crash escalation.

The exact procedure and results from the large scale attack cannot be presented in this master thesis due to its sensitive nature.

5.4 Results

The robustness testing of NAS showed some interesting result, where crashes were found. Since NAS was modeled completely in TTCN-3 and in full detail according to the standard, covering all 56 different message types in ESM and EMM, the full protocol could be generated by the fuzzer. The results presented here show statistics of the message types causing crashes, what

protocol they belong in (ESM or EMM) as well as a comparison between random fuzzing and our fuzzer. Note that the results presented in this thesis only consist of relative numbers and no absolute numbers will be revealed.

5.4.1 Message types statistics

To calculate statistics over the messages that caused crashes, the test case described in Section 5.2 was used which returns a list of message sequences causing a crash. The statistics are based on a large number of such sequences, with the total number of messages sent to create these sequences being of several factors larger. To make it possible to compare sequences easily, only sequences containing exactly one message was collected. The fuzzer picks messages at random and with a large number of messages sent, all 56 message types in NAS will be chosen many times with high probability.

The distribution of the number of crashes over the messages types, which were able to cause an Erlang worker to crash, can be seen in Figure 5.4. It shows the percentage of crashes per message type relative to the total number of crashes[1]. The graph shows clearly that the message type *O* has triggered more crashes than any other. From the presented statistics, some other results can also be illustrated.

A measurement of how many message types that were represented in the messages causing crashes was taken and the result can be seen in Figure 5.5. The total number of message types in NAS (combining both ESM and EMM message types) are 56, which means that only 15 message types were involved in causing crashes.

The results in Figure 5.6 show that there was a clear over representation of ESM messages among the crashes. Only two EMM message types were represented among the crashing message types, but these two stood for 17.9% of all crashes.

Distinguishing message that should, according to the NAS standard [26], originate from the UE versus the MME, the results in Figure 5.7 show that there were slightly more MME originating messages causing crashes than UE originating messages.

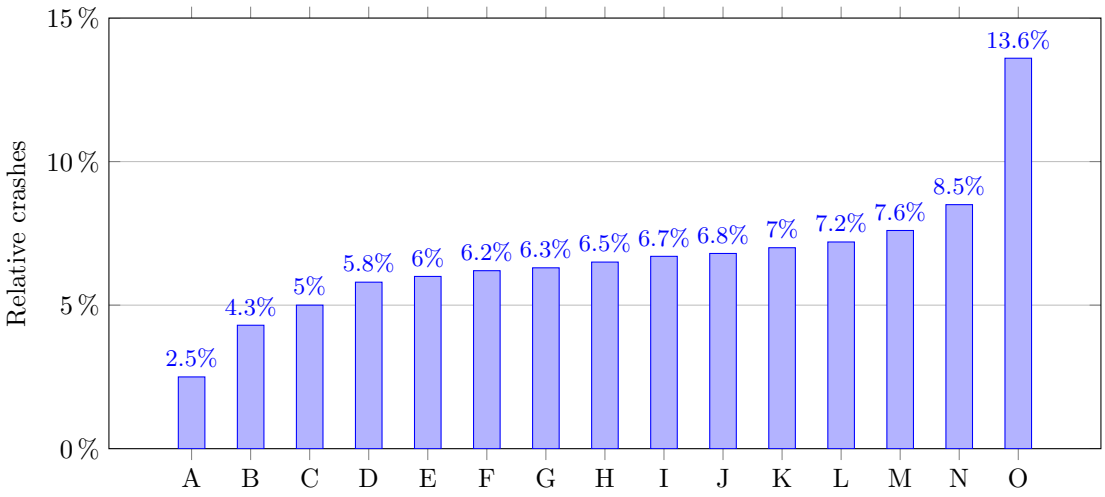


Figure 5.4: Relative number of crashes per message type

[1]The actual message types are replaced with identifiers from A to O

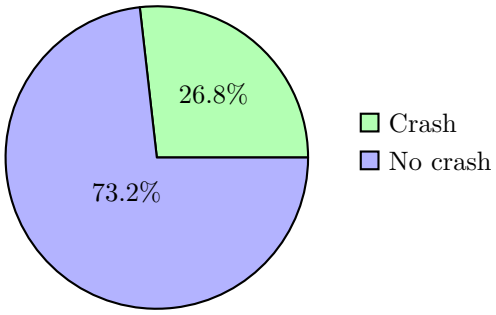


Figure 5.5: Percentage of message types causing crashes

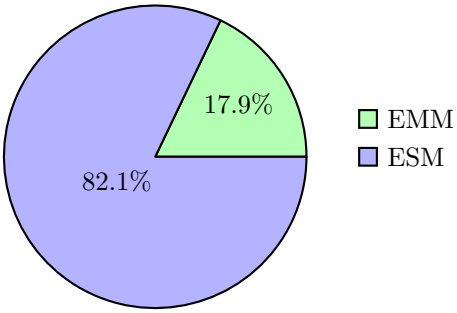


Figure 5.6: Percentage of ESM vs EMM messages causing crashes

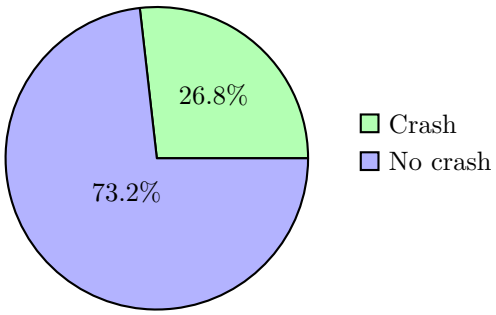


Figure 5.7: Percentage of messages that should originate from UE or from MME

5.4.2 Comparisons

To evaluate some choices made in designing the test case, as well as to compare the implemented fuzzer against different techniques, other test cases were created. A very simple random-based fuzzer was created, as described in Section 2.7.4. This had no mutation capabilities or any other type of way to more intelligently create messages, but was purely generating completely random amount of bytes, consisting of random data. The developed fuzzer, called *Model-based*, and the random-based fuzzer are compared in three different test cases in Figure 5.8. Each test case consisted of sending a very large number of messages each and the results consist of the number of occured crashes. Note that the actual number of messages sent to the SUT cannot be revealed

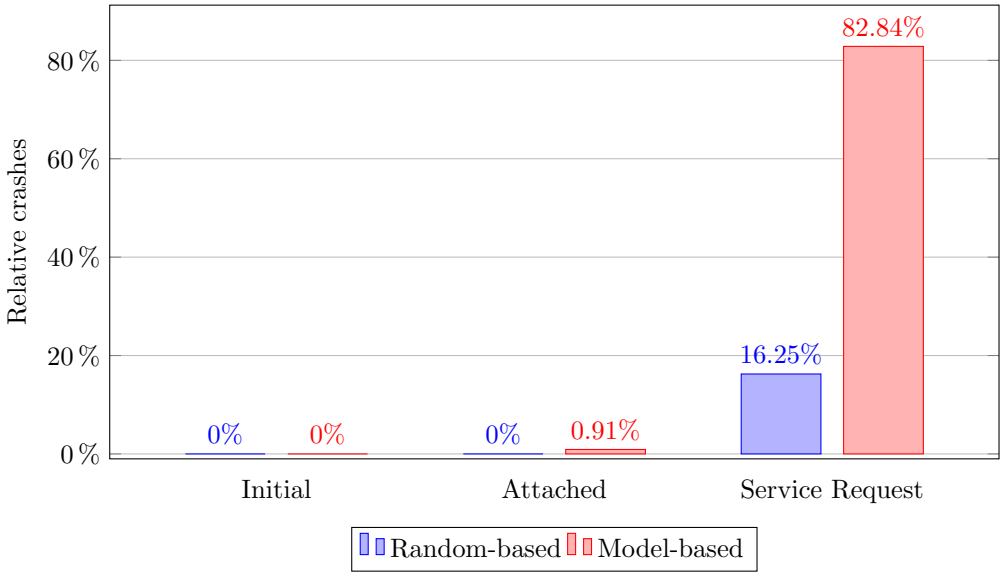


Figure 5.8: Comparison of crashes between random-based and model-based fuzzing, relative to the total number of crashes

because of its sensitive nature.

The first test case, called *Initial*, sent messages from the UE without attaching or performing any initial setup. This is the same way an *Attach Request* is sent when a UE wants to attach. The result shows that neither the random-based nor the model-based fuzzer could provoke any crash in this stage.

The second test case, called *Attached*, performed the attach phase as described in the real test case in Section 5.2 and then sent the generated messages when the EMM state was *EMM-REGISTERED*. As in the real test case, it made sure that the UE stayed in the wanted state by handling incoming messages. In contrast to the real test case however, it did not try to get in to some EMM mode, but assumed the mode to be *EMM-IDLE* all the time. The result show that the random-based fuzzer could not provoke any crash, and the model-based fuzzer could only provoke a small number of crashes compared to the *Service Request* test case.

The third test case, called *Service Request*, was the real test case as described in Section 5.2. The results show that the model-based fuzzer provoked more crashes while at the same time the random-based fuzzer actually provoked some crashes as well. In these results, the model-based fuzzer performed better than the random-based fuzzer with a factor of 5. Comparing the different test cases show that this combination of the EMM state *EMM-REGISTERED* and mode *EMM-CONNECTED* did outperform the others with a wide margin.

Another choice made in the real test case was to encrypt the messages when this was required in normal procedures. The assumption was made that more messages were to be allowed, and that only the decryption part of the SUT would be tested if encryption was required and not performed. The real test case, presented as *Service Request* in the previous Figure 5.8, was modified to disregard the encryption of messages, even when required. The results in Figure 5.9 shows the number of crashes produced after sending the same amount of messages as the previous comparison each for both the random-based and the model-based fuzzer, unencrypted and encrypted. These results show that while encrypted messages provoke more crashes, even with

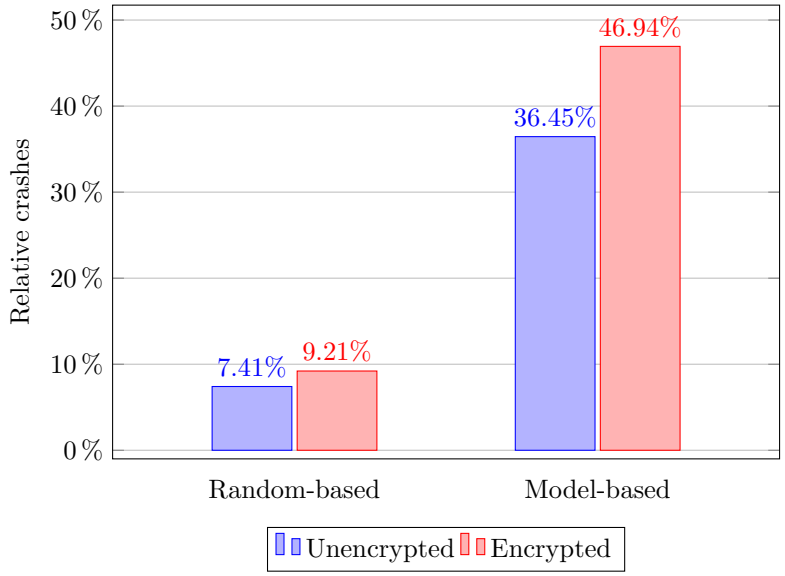


Figure 5.9: Comparison of crashes between unencrypted and encrypted messages, relative to the total number of crashes

the random-based fuzzer, the unencrypted messages did provoke crashes as well. Looking at the model-based fuzzer, the encrypted messages provoked almost 29% more crashes than the unencrypted messages. It is however important to notice that the number of crashes does not correspond to the number of actual bugs.

Chapter 6

Evaluation

The results presented in the case study showed that the fuzzer is capable of generating messages according to the extracted protocol models to test the system implementation. The observer monitored an Erlang crash counter, provided by the SUT, and reported the message which caused the Erlang worker to crash. Even though these Erlang crashes could be intentional crashes, the number of crashes indicates that some errors are likely to have been found. Therefore, it could be argued that the case study shows that the fuzzer is capable of finding faults that were previously unknown.

6.1 Observer Implementation

Fuzzing have great potential of finding not yet exploited bugs of system implementations, but an effective observer has to be implemented to reveal these unwanted behaviors created by the fuzzer. The implemented observer in Section 5.2 only monitored worker crashes to determine if generated message caused any faulty behavior. The worker crashes were considered to be the most suitable behavior to monitor, but several other properties were considered during the design phase which never was implemented.

Hardware utilization such as CPU usage and memory consumption was discussed to be monitored by the observer as a side track of what was considered to be a faulty behavior. A higher work load could mean that the worker have ended up in a state which it could not exit. The observer should look for CPU spikes to determine if such state has occurred. Higher memory consumption than what is considered to be acceptable for a single process would also be a measurement to consider. Both ideas could easily be implemented in the current observer. However, what is considered to be an acceptable value for both cases could not be established and the idea was abandoned.

Another property to observe would be to take advantage of the separated state machines in the UE and MME. The observer could trigger if the state machines in the UE and the MME believe they are in different states. An example of an erroneous state would be if the UE thinks that it is detached while the MME still believes the UE is attached to the network. This could potentially create a denial of service if enough UEs are capable of detaching while the MME thinks they still belong to the network.

The observer can also be state aware and always keep track of the latest entered state to determine if the current state is possible to reach from the previous state. Unfortunately, there was no built-in functionality to read the current state of a UE. The only information which can be obtained from the MME was if the UE was idle, active or in a transaction. This information

was not sufficient enough to determine if some erroneous state has been reached. A lot of work has to be done to keep track of the current state and what states it should be able to reach. This was considered to be too time consuming and out of scope of this thesis.

A technique often used with black-box testing is to set up a legitimate client (a UE using telecom terms) which does some repeated operation with the network. In this way an error can be detected by checking if the legitimate user still can communicate correctly with the system. This is useful to determine whether the SUT is still functional or has in some way crashed, when no other indication of this exists. Since we used a white-box approach this would not give use any further information beyond the information we can acquire from the system.

6.2 Test Execution of NAS Protocol Implementation

The case study presented in Section 5 gave us some interesting insight about the SUT. Because of the Erlang philosophy to handle errors by "letting it crash", it is not known how many of the malformed messages, which created a crash, were deliberate crashes or a system implementation bug. However, after a discussion with the test team it was concluded that at least one of the presented crashes was unintended. The crash logs have been handed over to the test team for further analysis.

This thesis does not cover the design of good test cases; instead it aims to provide experienced testers with a tool to generate fuzzed messages in an existing test environment. To get some information of the capability of the developed fuzzer, a case study was performed on a protocol implementation, in this case NAS.

The pseudo-code presented in Figure 5.1 was implemented in the test environment to find a number of unique messages which created an Erlang worker to crash. The test case was designed to test the system implementation of the NAS protocol. The purpose of this test case was to show that an erroneous behavior could be provoked by the implemented fuzzer. Even though all different message types of the NAS protocol were generated during the test run, only 15 out of the 56 message types, which corresponds to roughly 27%, were able to provoke a dynamic crash as can be seen in Figure 5.5. Two out of these 15 message types belonged to the EMM part of the NAS protocol, illustrated in Figure 5.6. There are in total 33 EMM and 23 ESM message types which imply that over 56% of the ESM messages and just over 6% of the EMM messages caused dynamic workers to crash in the developed test case. The generated messages sent to the SUT took no respect regarding the intended origin of a specific message type. Figure 5.7 illustrates the distribution of the collected messages of the intended origin according to the standards [26]. All messages were however sent from a UE to the MME. 43.5% out of the messages, which caused a crash, were supposed to originate from a UE while 56.5% messages were supposed to originate from the MME. The UE messages should be covered by the system implementation and would at least be accepted by the system implementation which implies that the randomized values in the data fields of the message probably caused the process to crash. The messages that should have originated from the MME however are not expected from the system implementation and it could be argued that this is the reason why MME messages are over represented in the sampled crashes.

6.3 Fuzzing Technique

To emphasize the efficiency differences of random and model-based fuzz testing, several tests were designed which are presented in Section 5.4.2. A large number of messages were generated and sent to the SUT with the same state awareness and encryption logic for both fuzzing methods

in all test cases. The first comparison is illustrated in Figure 5.8 and shows three different test cases. The first test case was not able to create any crash at all and no valuable information could be drawn from this test regarding the efficiency of random and model-based fuzzing in the current test environment. The second test showed that the model-based fuzzer were able to provoke a small number of crashes, while the random-based fuzzer did not find any messages at all which could cause a worker process to crash. However, the third test is what makes the model-based fuzzer to stand out from a less sophisticated fuzzer, such as a random fuzzer. The model-based fuzzer were able to provoke over five times more crashes for the same amount of generated messages, showing that a model-based approach is far more effective than a random approach. The random-based fuzzer was however capable of provoking worker crashes with almost no time spent on development. It could be argued if it is worth its time to create a more effective fuzzer when a basic is capable of finding worker crashes as well. A random-based fuzzer cannot test any sophisticated attack patterns which will decrease the coverage factor of the system implementation. Hence, a model-based fuzzer has great potential to find bugs in more complex systems. It's only the tester that sets the limit about how sophisticated the test execution will be.

A second test was developed to motivate the choice of using encryption of the generated messages sent by the fuzzer to the SUT. The random-based and model-based approaches were both tested with and without encryption which is illustrated in Figure 5.9. With encryption, both the random-based and the model-based fuzzers where able to provoke slightly more worker crashes compared with no encryption. The increase of crashes found with encryption turned on is approximately the same for both methods.

6.4 Large Scale Attack

Detailed results of the large scale attack are not discussed in this thesis due to its sensitive nature.

It could be argued that a large-scale attack is only possible in a controlled environment like the one it was tested in. To perform it in practice, the attacker needs to have a large number of UEs that could send crafted NAS messages. This may seem like two difficult problems to solve in practice, but below are possible solutions presented to both these problems that could make this attack seem possible.

The possibility for a user to send arbitrary messages relies on an open LTE stack that can be used to send messages of the NAS protocol. In Section 3.2, it is shown that an open GSM stack could be created and used to send arbitrary messages in the GSM network. The section also describes the increasing interests of the LTE network where the aim is to transmit and receive UE traffic. The technology to attack the LTE network exists, the question is not if, but when a serious attack against the LTE network will be carried out.

The possibility for a user to send messages from multiple UEs is either dependent on the number of UEs that the user possesses, or if there is some way to simulate several UEs. The NAS denial of service attack present in Section 3.1 describes a method to collect IMSI values. The attack uses the IMSI values to create a virtual UE per IMSI value, which gives a single device the possibility to emulate multiple UEs, dependent on how many IMSI values that could be collected. This is used to overload the system, but could instead be used to attach virtual UEs that all send the crafted NAS messages. As long as a sufficient amount of virtual UEs are attached, the attack presented in Section 5.3 could be launched.

6.5 Generalizability

The fuzzer is designed to be generic to any protocol modeled in TTCN-3, but the case study can only show that this was successfully applied by us with the NAS protocol. To further test its generalizability, the fuzzer was tested on the S1AP and Diameter protocol. Both these protocols are used in the MME, as described in Section 2.1.4, and are tested in the TTCN-3 conformance test environment.

The Diameter protocol was modeled in TTCN-3 and the fuzzer could extract this protocol fully. The protocol specification is very flexible and allows for custom extensions. The types included in the models were all specified in the Diameter specification [24] and extensions in 3GPP specification TS 29.272 [25]. No problems occurred for the model extractor, making the fuzzer capable of generating 100% of the modeled protocol. Using the MME as the SUT, Diameter messages are exchanged between the HSS and the MME. However; they are only sent from the HSS on demand of the MME as answers to requests. Therefore, the test logic requires that such requests could be created to get opportunities to answer with generated messages. Time was not put on developing such test logic and no test with sending generated Diameter messages was performed.

The S1AP protocol was not modeled in TTCN-3 but in ASN.1 which TITAN supports, and that presented some different problems for the model extractor. When running fuzzing tests, the fuzzer could generate 83% of all S1AP IEs. For example, the native data type `objid` was not supported by our fuzzer and workarounds had to be implemented for some TITAN quirks for the generated C++ files from the ASN.1 files. The messages generated were sent in TTCN-3 in the same way as with NAS with the help of the FAST framework.

The fuzzer exposed three main functions for S1AP message generation which corresponds to the three main message types of S1AP: *InitiatingMessage*, *SuccessfulOutcome* and *UnsuccessfulOutcome* [28]. When performing the basic fuzz testing on S1AP, only the *InitiatingMessage* type was generated and sent to the SUT since this was seen as the most complex type while the other were mere answer and acknowledgement messages.

However, testing S1AP may not be as interesting as NAS in respect to the level of trust of the input. The NAS messages are forwarded unmodified from the UE directly to the MME via the eNodeBs' S1-MME interface. The only level of trust in this case is the eventual assumption that the software generating the NAS messages is from a trusted developer, which is impossible to confirm. In the S1AP case, the messages are created by the eNodeBs and are at least originating from devices already trusted by the telecom network operator. A breach into the eNodeB software would of course make this untrusted.

6.6 Questionnaire

When the fuzzer had been designed, implemented, and tested, an evaluation of its usability and applicability of the test environment was performed within the organization. For this purpose we gathered the key testers of the MME to demonstrate the fuzzer and present one of the messages which provoked a crash while fuzzing the NAS protocol. The presentation included a live demonstration of how the fuzzer generated messages and that some of the messages were able to provoke a crash. The presented message, which caused an Erlang process to crash, was combined with the crash log with detailed information of the worker crash. After the presentation we asked the testers to fill in a questionnaire regarding the tool and the presented crash. Questions $\mathbf{Q_1}$ to $\mathbf{Q_7}$ are regarding the presented tool while $\mathbf{Q_8}$ to $\mathbf{Q_{11}}$ handles their thoughts about the presented crash. The answer to each question can be seen in Table 6.1.

Table 6.1: 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

Some of the answers span between 1 and 5 where 1 means "very unlikely" and 5 means "very likely". The following 11 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 crashes similar to the one presented? [Yes/No]

$\mathbf{Q_9}$: Would this tool save time in finding such crashes? [1-5]

$\mathbf{Q_{10}}$: What is the probability the crash would be found by you without this tool? [Percentage]

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

The answers provided us with interesting input about the usefulness and ease of use which can be seen in Table 6.1. Question $\mathbf{Q_1}$ got a rather negative response, in average 2.3 out of 5. One explanation could be that the questioned testers didn't perform this kind of negative testing in their daily activities, which means that no time could be saved. An average answer of 4.6 out of 5 on question $\mathbf{Q_2}$ shows that the testers think that the tool is capable of finding bugs which not would have been found with the current test methods. Question $\mathbf{Q_3}$ confirms with the average answer of 4.2 out of 5 that the quality of the product is believed to be improved if the tool would be used as a complement to the existing robustness testing already performed at Ericsson. The answers to Question $\mathbf{Q_4}$ (an average of 2 out of 5) and $\mathbf{Q_6}$ (1.75 days in median) show that the testers did not believe that they required to be experts in fuzzing to use the tool,

they thought it seemed easy to use and could be familiar with it after 1-3 days. The average answer 3.6 out of 5 on question $\mathbf{Q_5}$ indicates that the tool would fit with the existing testing requirements. Distinguishing the testing from positive and negative testing, the testers' average answer 4.17 out of 5 on question $\mathbf{Q_7}$ indicate that they believe the tool would be used.

The questions $\mathbf{Q_8}$ to $\mathbf{Q_{11}}$ relate to the crash presented for the testers. The average answer on $\mathbf{Q_8}$ indicates that similar crashes are rarely found. One third of the testers have unintentionally encountered similar crashes while testing other parts of the MME. This is further supported by the median answer of 10% on question $\mathbf{Q_{10}}$ where the presented crash would unlikely have been found early in the testing phase, if at all. The average answer 4 out of 5 on question $\mathbf{Q_9}$ states that the key testers of the MME believes that the tool would simplify the task of finding such crashes. Question $\mathbf{Q_{11}}$ gives an estimation of the time saved using the presented tool instead of finding the crash manually, given that a fault exists. By using the tool, the median value states that over 4.5 days could be saved.

6.7 Implementation Problems

Some problems arose due to our initial delimitations, see Section 1.2, and the design choices, see Section 4. In this section, the initial delimitations will be discussed and their impact of the results, as well as delimitations imposed by the design choices.

6.7.1 Initial Delimitations

The first and second delimitation states that the fuzzer will be specialized to work with the existing testing environment and the protocols tested there, not covering any other protocols or environments. The developed fuzzer is specialized to be integrated with TTCN-3, specifically with TITAN. It is however possible to perform fuzz testing on any protocol that can be tested with TITAN, which should be any protocol testable in TTCN-3. There are no other limitations regarding this and the case study, as well as other protocols tested described in Section 6.5, shows that this is possible.

The third delimitation states that the fuzzer will be limited in functionality in what the test environment and the chosen or developed fuzzing engine are capable of. The fuzzing engine was developed from scratch and an existing fuzzing engine was not used nor imported. Existing fuzzing engines, some presented in Section 3.4, provide features for generating values not just on random. Many of these features could be tasks for extending the developed fuzzing engine in the future, as is described later in Section 7.3.

The fourth and last delimitation states that potential erroneous behavior of the SUT will not be further investigated in this thesis, but left as a task for the developers and testers of the SUT. As is shown in the results of the case study, see Section 5.4, some crashes are believed to be faults that should be investigated and the developers are notified.

6.7.2 Imposed Limitations

There were some limitations imposed by the design choices that affected the functionality of the fuzzer. One limitation about the message generation was discovered with the integration of TITAN. During the case study, it was found that some messages did not match what the fuzzing engine generated, according to its logs. After investigating the messages sent using a network capturing tool, it was found that the message encoder overrides some generated values. While the selected message type and its contents were preserved, the NAS message type was overridden to match the message type selected in the TTCN-3 `union` field, as well as the protocol discriminator

field. This limits the number of fields that can be generated by the fuzzer, and it is impossible to know what fields without investigating the encoder. In the case of NAS, the fields that were discovered being overwritten would help the fuzzed messages being parsed more correctly since it made important values correspond to the TTCN-3 model. In other protocols, other fields may be overwritten which may prevent them from being tested by the fuzzer. It was deemed too time-costly to investigate exactly what fields that were affected, since that will depend on the protocol encoder.

6.7.3 Case Study Problems

When designing the case study, the FAST framework was chosen to integrate with, as mentioned in 4.1. While FAST had a lot of functionality that could be used, such as the attach phase in NAS, it was designed for positive testing and to fail fast. That means that if any unexpected message would arrive, the test case would stop and all progress was lost. This led to the necessity of reimplementing some functionality of FAST, mostly the protocol logic parts that involved waiting for answers. The functions which would actually create and send messages could be used to great extent.

The FAST framework was also designed to only support one single UE while testing the MME. All the simulated nodes were separated in TTCN-3 components, but the simulated UE only existed as information in a structure variable `vUE` global to all components. This means that the SGW, HSS and eNodeBs could read `vUE` and knew what to expect in arriving messages. If a message arrives that does not match the information stored in `vUE` the test case would end with a failure verdict. While this is helpful when designing positive testing for one UE, using several UEs are impossible without changes to the FAST framework.

For the large scale attack presented in Section 5.3, multiple UEs were needed. This led to designing a test case with reimplementing some of the UE logic from FAST. For each UE to simulate, `vUE` was copied in to an array `UEs[i]` and values such as the IMSI were changed to make the UE unique. The eNodeB functions could mostly be used but those functions reading directly from `vUE` had to be avoided and reimplemented. For the most parts, those functions were wrappers around a series of other functions that did not read `vUE`, which could be called directly instead. The SGW and HSS were reimplemented as very simple nodes, accepting every request coming in. Since the UEs would only attach and send the crashing messages, not much was needed to handle in the SGW and HSS. These limitations of FAST show that the framework is designed for positive testing. However, the large scale attack is not an ordinary fuzzing test case but a very special test case demanding more of the environment than the other fuzzing test cases.

Chapter 7

Future Work

The presented approach provides the basic functionality to apply robustness testing with model-based fuzzing. By extracting protocol models from an existing test environment, the fuzzer can generate messages from scratch. In this chapter, some possible extensions to the developed fuzzer are presented. They take different paths, but they are all based on interesting research and tend to get some more sophistication in regards to the generation of messages.

7.1 State Monitoring

This thesis focused on Erlang worker crashes and escalation of supervisor crashes where the state awareness has been limited to reach a state where it is possible to send valid NAS messages to the SUT. Another interesting aspect would be to monitor the state that the UE will reach depending on the messages sent to the SUT. Since the UE and the SUT have separated state-machines, they can be evaluated to see if any malformed message will make the two state-machines to have different views. The same fuzzing techniques can be applied to this as was presented in this thesis, except for the aim and monitoring decisions. Several questions arise with this approach:

- Is it possible to reach a state that the UE should not be able to reach?
- Is it possible to reach a state where harm can be done to the system or other subscribers?
- What could an attacker gain with state awareness?

7.2 Code Coverage

It was mentioned earlier in this thesis that with white-box testing the opportunity of getting high code coverage arises. This was not a part of this thesis but might be a worthwhile extension to implement in the future. A code coverage tool, such as an application that return the functions, or even lines of code, that are executed during fuzzing would show the coveage of the system code. The fuzzer can use this information to generate special messages that would hit the functions that are rarely accessed under normal conditions. Only the fields that will guarantee that these code paths will be hit can be fixed while all other fields are randomly generated. This procedure would preferably be fully automated.

7.3 Boundary Value Analysis

Messages in a textual protocol consists entierly of character strings. The various fields in the message then often have variable lengths, as compared to binary protocols like NAS which often have fixed lengths. When randomly generating messages for fuzzing, an unlimited number of strings can be composed to constitute a message. Therefore, it is obvious that more logic behind generation other than purely randomized values are needed, as done by Xu et al. [53] with different attack vectors.

When designing the fuzzer presented in this thesis, the NAS protocol was studied to get requirements of the fuzzer. When analyzing NAS, a large majority of the message fields were binary enumerations, like for example the EPS attach type field. This field is a three bit long field, where 001 means "EPS attach", 010 means "combined EPS/IMSI attach", 110 is "EPS emergency attach", and 111 is reserved. All other values are to be interpreted as "EPS attach" [26]. Interpreting the field as an integer and choosing a boundary value such as $MAX-1 = 110$ would only choose "EPS emergency attach" without meaning anything special. A random choice has more probability to choose interesting values such as 011.

However, some NAS fields are integers which are often length fields indicating how many elements the following array has. In such cases, a good extension to the fuzzer would be the ability to test inconsistency between actual array lengths and length field values. Sequence number fields and timer fields could as well be tested with boundary values which are more common to cause errors.

7.4 Detailed Message Field Analysis

It was found in the case study that randomly generated messages may cause an Erlang worker to crash, but the messages that caused a crash where not investigated in detail. Further investigation of message fields can be a future extension of this master thesis. One such extension could be to provide the fuzzer with mutation-based capabilities that systematically mutates fields of a known malicious message to find the field that caused the potential erroneous behavior (i.e. worker crash in the case study).

Several different strategies could be used to mutate the message. One strategy is to look at the optional fields in the TTCN-3 specification of the protocol. For every such field, the field could either be present or omitted. If the field is present, it contains a generated value. The mutator could toggle the presence; if the value was present, omit the value; or if the value was omitted, change it to present and generate a value for it. The effect of the toggling is measured after sending the new message with this field toggled. If the erroneous behavior caused by the first message is gone after toggling the field, the previous state of the field is deemed as important for triggering the behavior.

Another strategy for a non-optional field is to regenerate its value to see the effect with a completely new value. If the behavior is the same with the new message, this changed value is considered as not important for the effect caused in the first message. However, if the behavior of the first message is gone after modifying this field, the field is deemed as important.

Chapter 8

Conclusions

This thesis presents an approach for robustness testing using model-based fuzzing. Fuzzing is a software testing technique where malformed messages are created and sent to the system under test. Model-based fuzzing uses models of the message format to more intelligently create malformed messages. A tool for model-based fuzzing is developed which is fully integrated with an existing TTCN-3 testing environment. TTCN-3 is a standardized testing language and environment commonly used for conformance testing. The integration is made possible using TITAN, a TTCN-3 compiler and runtime environment developed by Ericsson. The tool makes use of networking protocol models used for conformance testing in TTCN-3 to perform model-based fuzzing.

With the integration of the tool in TTCN-3, any protocol modeled in TTCN-3 can be used for fuzzing. Since the tool is generation-based, every part of the protocol message structure can be created with every possible combination. This gives the opportunity to achieve good testing coverage. For an organization performing conformance testing with TTCN-3, this means that an evaluation of the robustness can be performed with good coverage.

The tool is evaluated with a case study of testing the 3GPP NAS protocol on the Ericsson SGSN-MME. Developed test cases make use of existing functionality in the environment to reach certain states before performing fuzzing. The test cases aim to find Erlang crashes which may not correspond to actual bugs, but being an indication of where to investigate further. Results of the testing show that it is crucial to perform fuzzing in certain states of the protocol. The tool is further compared to a random-based fuzzing approach, which shows that model-based fuzzing performs better in terms of causing several more crashes.

By conducting a questionnaire with key testers of the Ericsson SGSN-MME, opinions about the fuzzer's usability is obtained. Among the testers, the fuzzer is considered easy to use and that it would increase the quality of the product tested with it. It is also considered to be a valuable tool which would take short time to be familiar with. The testers also believe that the fuzzer can find crashes that would not be found without the tool.

Abbreviations

1G First Generation

2G Second Generation

3G Third Generation

3GPP Third Generation Partnership Project

3GPP2 Third Generation Partnership Project 2

4G Fourth Generation

AP Application Processor

API Application Programming Interface

ASN.1 Abstract Syntax Notation One

BS Base Station

CDMA Code Division Multiple Access

CDMA2000 Code Division Multiple Access

CH Component Handler

CMXB3 Component Main Switch Board version 3

CPU Central Processing Unit

DP Device Processor

E-UTRAN Evolved Universal Terrestrial Radio Access

ECM EPS Connection Management

EMM EPS Mobility Management

eNodeB Evolved Node B/ E-UTRAN Node B

EPC Evolved Packet Core

EPS Evolved Packet System

ESM EPS Session Management

ETS Executable Test Suite

ETSI European Telecommunications Standards Institute

EU European Union

FSB File Serving Board

GEP3 Generic Ericsson Processor board version 3

GPRS General Packet Radio Service

GSM Global System for Mobile Communications

GTPv1 GPRS Tunneling Protocol version 1

GTPv1-U GPRS Tunneling Protocol User Plane version 1

GTPv2-C GPRS Tunneling Protocol Control Plane version 2

GTT GSN Test Tool

HSS Home Subscriber Server

IEEE Institute of Electrical and Electronics Engineers

IMSI International Mobile Subscriber Identity

IMT-2000 International Mobile Telecommunications-2000

IMT-Advanced International Mobile Telecommunications-Advanced

IP Internet Protocol

LTE Long Term Evolution

MME Mobility Management Entity

MMS Multimedia Messaging Service

MS Mobile Station

NAS Non-Access Stratum

NCB Node Control Board

NMT Nordic Mobile Telephone

OpenBSC Open Source Base Station Controller

Osmocom Open Source Mobile Communications

OsmocomBB Open Source Mobile Communications Baseband

PDN Public Data Network

PGW PDN Gateway

PIU Plug-In Units

PSTN Public Switched Telephone Network

QoS Quality of Service

RAN Radio Access Network

RFC Request For Comments

RRC Radio Resource Control

S1AP S1 Application Protocol

SCTP Stream Control Transmission Protocol

SCXB2 System Control Switch Board version 2

SGSN Serving GPRS Support Node

SGSN-MME Serving GPRS Support Node – Mobility Management Entity

SGW Serving Gateway

SIP Session Initiation Protocol

SMS Short Message Service

SSD Solid State Disk

SUT System Under Test

TA Tracking Area

TCP Transmission Control Protocol

TE TTCN-3 Executable

TTCN-3 Testing and Test Control Notation version 3

UDP User Datagram Protocol

UE User Equipment

UMB Ultra Mobile Broadband

UMTS Universal Mobile Telecommunications System

USB Universal Serial Bus

WCDMA Wideband Code Division Multiple Access

WiMAX Worldwide Interoperability for Microwave Access

List of Figures

List of Tables

Bibliography

[1] I. T. 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

[2] J. Moteff, C. Copeland, and J. Fischer, "Critical infrastructures: what makes an infrastructure critical?" DTIC Document, 2003. [Online]. Available: http://www.fas.org/irp/crs/RL31556.pdf

[3] 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

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

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

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

[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. [Online]. Available: https://www.usenix.org/legacy/event/sec11/tech/full_papers/Mulliner.pdf

[9] R.-P. Weinmann, "All your baseband are belong to us," Presentation, DeepSEC, 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] 3GPP, "About 3GPP," 2013. [Online]. Available: http://3gpp.org/About-3GPP

[13] GSM Association, "GSM technology," 2013. [Online]. Available: http://www.gsma.com/aboutus/gsm-technology/gsm

[14] Qualcomm, "History - Who We Are," 2013. [Online]. Available: http://www.qualcomm.com/about/history/

[15] G. L. Stüber, *Principles of mobile communication*. Springer, 2011.

[16] *Framework for services supported by IMT*, ITU-R Std. M.1822, 2007. [Online]. Available: https://www.itu.int/dms_pubrec/itu-r/rec/m/R-REC-M.1822-0-200710-I!!PDF-E.pdf

[17] International Telecommunication Union, "ITU paves way for next-generation 4G mobile technologies," 2010. [Online]. Available: http://www.itu.int/net/pressoffice/press_releases/2010/40.aspx

[18] J. Gozalvez, "First Commercial LTE Network [Mobile Radio]," *Vehicular Technology Magazine, IEEE*, vol. 5, no. 2, pp. 8–16, 2010.

[19] Ericsson AB, "CPI," [Internal Ericsson].

[20] 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

[21] 3GPP, "3GPP TS 36.331 V11.4.0: Radio Resource Control (RRC)." [Online]. Available: http://www.3gpp.org/ftp/specs/html-INFO/36331.htm

[22] 3GPP, "3GPP TS 29.281 V11.6.0: General Packet Radio System (GPRS) Tunnelling Protocol User Plane (GTPv1-U)." [Online]. Available: http://www.3gpp.org/ftp/specs/html-INFO/29281.htm

[23] 3GPP, "3GPP TS 29.274 V11.7.0: Evolved General Packet Radio System (GPRS) Tunnelling Protocol for Control Plane (GTPv2-C)." [Online]. Available: http://www.3gpp.org/ftp/specs/html-INFO/29274.htm

[24] P. Calhoun, J. Loughney, E. Guttman, G. Zorn, and J. Arkko, "RFC 3588: Diameter Base Protocol," Internet Engineering Task Force, Sep. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3588.txt

[25] 3GPP, "3GPP TS 29.272 V11.7.0: MME and SGSN related interfaces based on Diameter protocol." [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/29272.htm

[26] 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

[27] 3GPP, "3GPP TS 24.007 V11.0.0: Mobile radio interface signalling layer 3." [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/24007.htm

[28] 3GPP, "3GPP TS 36.413 V11.4.0: Evolved Universal Terrestrial Radio Access Network (E-UTRAN); S1 Application Protocol (S1AP)." [Online]. Available: http://www.3gpp.org/ftp/specs/html-INFO/36413.htm

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

[30] ETSI, "201 873-1 Part 1: TTCN-3 Core Language, Version 4.5.1," 2013. [Online]. Available: http://www.etsi.org/deliver/etsi_es/201800_201899/20187301/04.05.01_60/es_20187301v040501p.pdf

[31] 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

[32] 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

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

[34] *Abstract Syntax Notation One ASN.1: Specification of basic notation*, ITU-T Std. X.680, 2008. [Online]. Available: http://www.itu.int/ITU-T/studygroups/com17/languages/X.680-0207.pdf

[35] J. Armstrong, "Making reliable distributed systems in the presence of software errors," Ph.D. dissertation, KTH, 2003. [Online]. Available: http://www.erlang.org/download/armstrong_thesis_2003.pdf

[36] J. Armstrong, "Concurrency Oriented Programming in Erlang," *Invited talk, FFG*, 2003. [Online]. Available: http://ll2.ai.mit.edu/talks/armstrong.pdf

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

[38] Ericsson AB, "Ericsson SGSN-MME." [Online]. Available: http://www.ericsson.com/ourportfolio/products/sgsn-mme

[39] I. Schieferdecker, J. Großmann, and M. Schneider, "Model-based fuzzing for security testing," 2010. [Online]. Available: http://www.spacios.eu/sectest2012/pdfs/SecTestICST_Schieferdecker.pdf

[40] K. V. Hanford, "Automatic generation of test cases," *IBM Systems Journal*, vol. 9, no. 4, pp. 242–257, 1970.

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

[42] A. Takanen, "Fuzzing: the Past, the Present and the Future," in *Actes du 7ème Symposium sur la Séurité des Technologies de l'Information et des Communications*, 2009.

[43] J. Neystadt and Microsoft Corporation, "Automated Penetration Testing with White-Box Fuzzing," 2008. [Online]. Available: http://msdn.microsoft.com/en-us/library/cc162782.aspx

[44] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "Finding software vulnerabilities by smart fuzzing," in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. IEEE, 2011, pp. 427–430.

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

[46] Codenomicon Ltd., "Fuzzing challenges: Metrics and Coverage," 2010. [Online]. Available: http://www.codenomicon.com/resources/whitepapers/codenomicon-wp-fuzzing-metrics-20100202.pdf

[47] D. Yu and W. Wen, "Non-Access-Stratum Request Attack in E-UTRAN," in *Computing, Communications and Applications Conference (ComComAp), 2012.* IEEE, 2012, pp. 48–53.

[48] OsmocomBB project, "OsmocomBB mobile." [Online]. Available: http://bb.osmocom.org/trac/wiki/mobile

[49] OsmocomBB project, "OsmocomBB Hardware/Phones." [Online]. Available: http://bb.osmocom.org/trac/wiki/Hardware/Phones

[50] D. Spaar, "Running your own LTE eNodeB." [Online]. Available: http://www.mirider.com/weblog/2013/07/30/

[51] R. Amin, "4G Wireshark Dissector based on Samsung USB stick." [Online]. Available: http://labs.p1sec.com/2013/08/18/4g-wireshark-dissector-based-on-samsung-usb-stick/

[52] B. Wojtowicz, "OpenLTE." [Online]. Available: http://openlte.sourceforge.net/

[53] 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.

[54] 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.

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

[56] 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

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

[58] P. Amini, "Sulley," 2013. [Online]. Available: https://github.com/OpenRCE/sulley

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

[60] Codenomicon Ltd., "Codenomicon Defensics Traffic Capture Fuzzer," 2013. [Online]. Available: http://www.codenomicon.com/defensics/traffic-capture-fuzzer/