POLITECNICO DI TORINO & EURECOM - TÉLÉCOM PARIS

Master Degree Course in Computer Engineering

Master Degree Thesis

E-WASSI: Evolutionary Worldwide Application for System Security and Information

Supervisors
prof. Marc Dacier
prof. Guido Marchetto

Assistant supervisor
prof. Fulvio Valenza

Candidate
Antonino Vitale

Academic Year 2020-2021

To my parents Santina and Francesco, who always supported me and every decision I took

To my sister Simona, for being always by my side despite the thousands of fights we had

To my grandparents, who always protected me, even from the heaven

To my childhood and teenage years friends, for our bond which gets stronger even if we are far away

To my friends from Turin, for giving me a shelter and hospitality when I needed it

To my "portugese" friends, who made me experience one of the best period of my life

To my Eurecomian

friends, for having shared
good and bad moments
but always with the smile

Abstract

English version

Nowadays, the Internet of Things and Operational Technology worlds are evolving thanks to the introduction in such environments of the IP protocol, which brings several benefits but at the same time has the main drawback of increasing the attack surface. Since the security of the devices belonging to those worlds is governed by security elements such as firewalls, NATing devices and IPS, E-WASSI proposes the introduction in these environments of components called *the Guardians* communicating each other through the use of a new-defined protocol called, precisely, *WASSI* with the goal of assessing and protecting such devices. The Guardians will leverage existing solutions in the context of network security and on top of that it will provide as a novelty functionalities to verify that those devices are acting as supposed. The role played by WASSI is to define packets sent or received by the components involved in a test campaign. This thesis focuses on the development and implementation of some test cases that will allow the Guardians to speak the WASSI protocol. The test cases are developed using TTCN-3, a technology specifically designed for testing and verification with the support of Titan which provides a compilation and execution environment.

Version française

De nos jours, les mondes des Internet of Things et des Operational Technology évoluent grâce à l'introduction dans de tels environnements du protocole IP, qui apporte plusieurs avantages mais en même temps a le principal inconvénient d'augmenter la surface d'attaque. Depuis que la sécurité des appareils appartenant à ces mondes étant régie par des éléments de sécurité tels que les firewall, les appareils de NATing et IPS, E-WASSI propose l'introduction dans ces environnements de composants appelés *the Guardians* communiquant entre eux grâce à l'utilisation d'un nouveau protocole appelé, précisément, *WASSI* dans le but d'évaluer et de protéger ces appareils. The Guardians exploitera les solutions existantes dans le contexte de la sécurité du réseau et en plus de cela, il fournira en tant que nouveauté des fonctionnalités pour vérifier que ces appareils agissent comme prévu. Le rôle joué par WASSI est de définir les paquets envoyés ou reçus par les composants impliqués dans une test campaing. Cette thèse se concentre sur le développement et la mise en œuvre de certains test cases qui permettront aux Guardians de parler le protocole WASSI. Les test cases sont développés en utilisant TTCN-3, une technologie spécialement conçue pour les tests et la vérification avec le support de Titan qui fournit un environnement de compilation et d'exécution.

Acknowledgements

The work presented in this thesis has been developed under the supervision of:

- Prof. **Marc Dacier**, supervisor for EURECOM
- Prof. **Guido Marchetto**, supervisor for Politecnico di Torino
- Prof. **Fulvio Valenza**, assistant supervisor for Politecnico di Torino

A special thanks go to Prof. Marc Dacier for having given me the possibility to work on this project and for the attention that he put on me from the first to the last day, complemented by his advice and patience.

Contents

List of Figures

Chapter 1

Introduction

1.1 Context

The cities we are living in are evolving day by day with respect to the evolution and the spread of the technology. Most of the things we can find around us are likely to be connected to the Internet or, at least, it is possible to find an object with the same purpose which implements that capability.
Home automation is one of the most representative examples: you can connect your own house to the Internet and control your heating system, windows and doors locks, television, lights and a myriad of other object through your smartphone. Domotics is not the only example, we can think also about the wearable devices such as smartwatches, fitness bands and even cars are connected to the Internet.
All the aforementioned objects can be grouped in what we call *Internet of Things* (IoT) world. It is not the only world where devices that were not meant to be connected on the Internet are nowadays growing in number. The *Operational Technology* (OT) [1, 2] world, i.e. the Industrial Control Systems (ICS) environment, was meant to include a relative small group of people and operators who could trust each other. In the last few years IP networks pervaded this world because of the elevated number of benefits that it brought out. In facts thanks to that network is possible to improve the efficiency, collect data to be analyzed and even monitor and control remotely.
The increasing number of such devices has an important drawback: the more devices are connected to the Internet, the bigger the attack surface becomes for malicious people. That drawback is also due to the poor attention the manufactures put when they design their devices in security point of view. The key words leading this environment are *first to the market*, then manufactures prefer, naively, releasing a product which presents security issues and vulnerabilities in order to be the firsts into the market to sell this new kind of smart object. The risks that are carried out by these vulnerabilities do not affect only the infected device but could be a threat even for others. These devices can participate to large Distributed Denial of Service (DDoS) attacks or other misdeeds as described in [3–6]. The targets can be OT environments as well such as critical infrastructures or plants, as happened with Stuxnet [7].

Some solutions to mitigate and reduce the attack surface already exists: NATing devices, firewalls and Intrusion Prevention Systems (IPS), Intrusion Detecting Systems (IDS), Network Access Controls (NAC), Data Leak Protection (DLP). The security of those devices belonging to the IoT and OT environments depends on the reliability of the aforementioned solutions. At this point E-WASSI takes place.

1.2 The Guardian

E-WASSI proposes the introduction in such environments of software components called **Guardians**. The Guardian's role is firstly to continuously verify that all the elements due the protection of the environment are correctly operating. Not only, the Guardian aims to analyze, assess and protect the network where it is located in, uncovering suspicious network elements with anomalous behavior. It does not rely on the information provided by the system because those sources could be compromised. It learns from the environment and adapts its behavior. The Guardians do not work stand-alone but they need each other to fulfill their mission, being part of a simple, evolutionary distributed architecture. Moreover, a Guardian guards the other Guardians as well, in order to check the integrity of each other. They report to a **Master node** in such a way to have a hierarchical deployment and simple management. The Master node will run on a private machine or in the cloud.
What the Guardians aim to can be resumed with the acronym **GUARD**:

- **G**ather information about the environment they have been deployed in
- **U**ncover network elements threatening the integrity or confidentiality of network packets
- **A**ssess continuously the security of the identified elements
- **R**ecognize any sign of ongoing intrusion or malicious behavior
- **D**ivert the attention of the perpetrator of an ongoing intrusion

Considering that the Guardian is a software solution, it can be deployed and run on any kind of already existing machine. In any case it is preferred to have it implemented on a cheap Raspberry Pi for security and simplicity reasons. In such a way it is possible to have a secure platform and make its deployment trivial for the users. Moreover, since the Guardians aims to protect all kinds of environments with their network protocol such as Bluetooth, Ethernet, 3G/4G/5G, Zigbee and so on, the Raspberry Pi support and can handle them all.
The simplest case where a Guardian can set in is a Local Area Network (LAN) with several IoT devices and their Internet access is ruled through a NAT, a firewall and an IPS. In such a LAN the Guardian covers different roles:

- Ensure that NAT, firewall and IPS are well configured and working properly

- Detect the presence of another active component on the path to the Internet
- Verify that no attacker has subverted another Guardian

1.3 WASSI Protocol

The last three points of the previous section bring out some very important questions: how can the Guardians succeed in their mission? Which are the resources at their disposal? The answer to these question is a very novel protocol used by the Guardians and the Master node to exchange information and communicate each other: **WASSI**.
WASSI is an application-layer protocol which defines packets that are sent and received between two Guardians selected from a pool of waiting hosts by the Master node. When two Guardians are chosen by the Master node, thanks to the information that this latter provides to them, they starts transmitting packets presenting a well-defined structure in order to identify the presence of other elements in the path that links them. A Guardian is able to perform this analysis comparing the packets it has sent to the packets received from the end-point. A very trivial example can be to compare the IP address a Guardian is claiming to have with the actual value available in the packet in order to discover if the Guardian is behind a NATing device.
The sequence of packets exchanged by two Guardians to answer a given question is called **test case**. The execution of other test cases may be triggered as subsequent answer to the result of a previous test case. In this way we create a chain of test cases required to qualify the connection between two Guardians. This collection of test cases is called **test case campaign** (or simply campaign). The campaigns are implemented thanks to an Abstract Test Suite called Test and Test Control Notation version 3, also known as TTCN-3, and Titan that is the Means of Testing allowing to compile the modules written using the language provided by TTCN-3 into executables, namely the so-called Executable Test Suite. The latter is composed by several modules with their own test cases that, once executed, will trigger the Guardians message exchange, that is the WASSI protocol.

1.4 GUARD

As aforementioned, the Guardians' goal can be summed up with the term *GUARD*. Each letter represents a different mission to be accomplished by the Guardians. We will analyze each of them in detail to have a better perspective on what and how the Guardians are due to do.

1.4.1 Gather

The main goal of this mission is to *gather (passively) information about the environment they have been deployed in.* Since they cannot rely on the information provided be the devices they are trying to watch and guard because they could be compromised, they should act passively without asking any kind of information to these last. This is a delicate task because of the fragility of IoT and OT devices due to their vulnerabilities.

In fact, even a simple and innocuous port scan or remote fuzzing can cause malfunction and, in the worst case, break them [8].
The best approach to be chosen in this case is using the so called **ZeroConf protocols** [9] whose development and spread is continuously increasing. These protocols allow to gather a huge amount of information about the devices located in the same local network by listening to their announcements and exploiting the broadcast and multicast requests. The most widely diffused of these protocols include mDNS (RFC6762 [10]), LLMNR (RFC4795 [11]), WS-Discovery [12], DNS-SD (RFC6763 [13]), SSDP [14], DLNA [15], ALLJOYN [16].

1.4.2 Uncover

Another important goal of the Guardians is *uncover network elements threatening the integrity or confidentiality of network packets*. This task can be carried to accomplishment thanks to the test case campaigns aforementioned. This is possible because the Guardians involved in a test case know exactly what it has been sent and received and, by comparing them, they can realize if the communication has been altered. A very trivial example to discover if a Man In The Middle (MITM) Attack is ongoing and attacking a secure, encrypted channel is to verify that the payload sent by a first Guardian is the the same of the one received by the second Guardian. If someone is playing the role of MITM the two payload must differ because the attacker has to decrypt the sent payload if he/she wants to have access to the plaintext and then it must be re-encrypted using a different key from the original one, causing the forge of a different payload. This effect is not specific only in this scenario but can be applied to other networking elements such as firewalls, NATing, DPI and so on.

1.4.3 Assess

The campaigns will be used as well to *assess continuously the security of identified elements*. Through a packet exchange it is possible to verify that those security elements protecting the local network are properly working. For instance we could verify that a firewall has correctly configured the NAT, or that its rules regarding ports, established connections, IP address filtering and so on are bringing out the desired outcomes.

1.4.4 Recognize

This mission aims to *recognize any sign of ongoing intrusion or malicious behavior*. What has to be noticed is that the aim is not to copy an already existing network based intrusion detection system. By leveraging the existing works we want to address the issue to provide network flow policy definitions. Starting from the identification step of the devices located in the local network, a Guardian builds a profile about each device of the expected network flow policy, by identifying the protocols they are supposed to talk, giving an approximation of the bandwidth, packet rate and so on. In such a way once that a device has been identified, it will be associated to these policies. Thanks to the latter, a Guardian can instantiate a model based also on the environment to verify that a device is not compromised and generate a normal behavior model that will be the reference point to detect a sign of infection. The main difference with the previous works which mostly failed is that the model is based on small devices whose network access is restricted to few functionalities whereas in the IT world the variability of the the network profiles to be considered is elevated.

1.4.5 Divert

The last mission of the Guardian, which a consequence of the the previous one, is to *divert the attention of the perpetrator of an ongoing intrusion.* As it has been explained before the IoT and OT devices are characterized by the usage of ZeroConf protocols which sends broadcast and multicast packets to find out the available services inside their network. In such a way an attacker can passively scan a network and easily identify the devices and, therefore, their vulnerabilities. The Guardian can exploit this type of communication to create a fake virtual device which will behave exactly as real device in order to attract the interest of an attacker. Once that an attacker has contacted our honeypot, the Guardian can profile its behavior providing valuable forensics information and eventually slowing down the bots. The forensics information will be collected by analyzing the responses to erroneous CRC, out-of-order packets, incorrect syntax or time out. The Guardians will generate alerts, thanks to MISP [17], to be consumed by security platforms which will proceed to the mitigation phase.

Chapter 2

State of the Art

This chapter includes two distinct sections: Related works and Tools. The former introduces the works existing in literature addressing the same problem of E-WASSI or leveraging systems and architectures that can be a base point for our work. On the other side, the latter presents some of the most known tools in the context of network security that could be involved in E-WASSI in order to fulfil its missions.

2.1 Related works

The work of [18] presents, after having introduced the several risks that threaten the security of IoT devices, the IoT security solutions in the existing literature. It distinguishes several types of solutions based on the technique used:

- **Blockchain**: all the feature that belong to the blockchain [19] make them perfect tool to develop a strong, tamper-proof, distributed and open data structure for IoT data. As regards the privacy of the user data stored in the IoT devices, a permissioned blockchain can secure the IoT devices as explained in [20–22]. The data manipulation is mitigated since thanks to the blockchain the devices are interlocked and then if any device modifies some data the system rejects it [23, 24]. The decentralized architecture of the blockchain reduces the cost and the traffic since the devices can directly communicate each other without passing through servers [25–27] and the heavy load that is on the cloud and the servers is distributed in different nodes of the network that are updated simultaneously eliminating in such a way single points of failure that could make the cloud services unavailable [28, 29].
- **Fog Computing**: with fog computing is meant the extension of cloud computing and services to the edge of the network, creating a 3-layer architecture composed by device, fog and cloud layers. Fog computing aims to enhance security, decrease the amount of data stored in the cloud and increase the overall performance of the IoT application. The data is collected by the so-called fog nodes which are devices that can be installed anywhere. Those nodes are inter-located between the IoT devices and the cloud, working as a sort of filter for the data directed to cloud that

is managed and preprocessed and can be stored in a sub-storage in the fog node. The work of [30] shows how is possible to implement a real-time service that can detect an attack on the cloud, whereas how is possible to implement a secure identity authentication system can be found in [31–34]. The data temporarily stored in the sub-storage must be encrypted using different keys for different owners. In order to enable data sharing in such scenario, [35] proposes a some cryptographic techniques such as key-aggregate encryption, proxy re-encryption and attribute-sharing. For bypassing the problem of searching on encrypted data, a search-able encryption is proposed in [36] and a dynamic symmetric search-able scheme is introduced in [37]. The fog nodes provide decentralized (secure) computation; for instance [38] presents a method called server-aided computation in order to achieve this goal.

- **Machine Learning**: Machine Learning applied to IoT security is a solution that has been taking place in the past years providing defence to attacks such as DoS [39, 40], eavesdropping [41, 42], spoofing [43–47], privacy leakage [48, 49].

- **Edge Computing**: edge computing introduces in a network composed by edge devices, fog nodes and the cloud the presence of edge servers between the devices and the fog nodes in order to perform processing activities and reducing the computational load of fog/cloud. The main advantages of edge computing are the fact that the data will be stored and processed within the device or local network, avoiding data breaches and the real-time response provided since the computation is performed physically near to the device.

The work of [50] proposes an hierarchical and distributed architecture to mitigate the workload and rule complexity problems in security control. The architecture is composed by several nodes with different scopes and goals. The regular traffic is analyzed by a cluster of IDS monitor nodes in order to balance the traffic between them. Each IDS node can detect an intrusion, creating an alert as feedback which will be forwarded to a Business Rule Engine (BRE) cluster that will perform a deeper analysis. The business rules consist of a map function and a reduce function that will firstly analyze the data coming from the IDS nodes and then the map output will be aggregated. This operation is performed by several intermediate BRE nodes, whereas the root will reduce the results coming from them to have a complete perspective of the system state. When a series of alerts is identified as an attack by the BRE, a proper response will be generated.

A model-based framework based on Attack Trees and Price Timed Automata for IoT system security in Smart Cities is proposed in [51]. Starting from model corresponding to the behavior of an attacker trying to threaten a system, an attack-tree [52] is created. This tree is, then, translated to a into a network of price timed automata [53] which on its side will be used to generate TTCN-3 test suites thanks to a test generation algorithm. The produced test cases will be executed by a cloud-based testing architecture that will collect and analyze the results.

At last, [54] proposes a distributed SDN security framework built on a multi-level remediation mechanism called Tennison. Tennison offers a multi-level capability for lightweight monitoring across a large number of flows and the possibility of performing DPI on a series of selected flows. Its architecture is composed by different layers: the low layer is called Collection Layer and includes instances of Bro (whose current name is Zeek [55]) and Snort [56] DPI, ONOS Controllers [57] and sFlowRT [58] which provide control and monitoring for higher layers. These istances are distribute all over the network in order to monitor different blocks. The data generated by the Collection Layer feeds the higher layer called Coordination Layer which is composed by the Tennison Coordinator and aggregates and stores all the information. The Coordinator, then, can control and modify the forwarding plane of the network by communicating with the ONOS Controller. The highest layer called Application Layer provides security applications that by interacting with the Coordinator realize custom and dynamic security behaviors in order to address specific threats by means of the creation of tailored application or leveraging existing tools.

2.2 Tools

2.2.1 AmpPot

AmpPot [59] is novel honeypot which aims to stop the so-called amplification attacks that are designed to overwhelm the targets with massive DoS attacks. Since many UDP-based protocols such as DNS, NTP, SSDP present vulnerabilities that could lead to such attacks, AmpPot tries to replicate those vulnerable protocols in order to attract attackers and profile the behavior of the malware to monitor techniques and targets. In such a way AmpPot tries to identify the source of amplification of the DoS attacks that is anonymous since it uses spoofed IP addresses. The technique used by AmpPot is to link the scanning and attacking phases of the attack in order to have several pairs that associate the scanner to the attack. Moreover, AmpPot tracks traffic patterns in order to map the responsible booter services.

2.2.2 Honeytrap

Honeytrap [60] is a framework for running, monitoring and managing honeypots. It is possible to develop complex honeypot architectures or a simple, single server. It can listen to all the ports in order to verify which are the ports at risk and to collect information or it is possible to listen to a specific port and give predefined answers. Furthermore, it is possible to forward and converge all the attack traffic on a the main Honeytrap.

2.2.3 nmap

nmap [61] is open-source tool for scanning and collecting information about the network in order to find out hosts and services running into the the network by sending packets and listening to their response. The functionalities provided by nmap are several, including operating system detection, port scanning, application version detection, auditing device and firewall security, response analysis, vulnerability exploitation.

2.2.4 ntop

ntop [62] is a software that tries to emulate what the command *top* does with processes but in a network perspective. In such a way it is possible to show the network usage and status. The program can display the hosts currently using the network and the traffic generated and received by each of them. Its functioning is based on the layer 2 and 3 (i.e. on Ethernet and TCP/IP) in order to have a complete network activity.

2.2.5 Zeek

Zeek [55] is an intrusion detection system with functionalities for analysis of network events. The core of the software is an event engine to whom the packets captured with pcap are delivered. The event engine decides if the packets are to be accepted or discarded and the ones that pass this selection are forwarded to a policy script interpreter. Those scripts are written in the Turing complete Zeek scripting language and are the handlers of the events that create the corresponding action policies. Moreover, Zeek provides several analyzers for application layer decoding, anomaly detection, signature matching and connection analysis.

2.2.6 Manufacturer Usage Description (MUD)

MUD (RFC8929 [63]) aims to provide a medium for end hosts to claim which kind of network functionalities the device requires in order to reduce the attack surface to only the protocols that it is supposed to talk with. MUD provides URLs that identify and describe the related network devices, whose description can be retrieved by the local network management systems. The description is encoded in form of YANG-based JSON files called namely MUD files. Those files are stored in a MUD file server which can be queried by a MUD manager, that is the network management system, thanks to the MUD URLs that refer to the MUD files.

2.2.7 Internet Motion Sensor (IMS)

IMS [64] aims to profile and track on world wide Internet. It is composed of topology-aware dark IP sensors and aggregators that are located in different, key locations in order to have a global perspective. Those sensors monitors different blocks of address space that can be a single host or even bigger networks as 8-prefix length subnets. By monitoring the blocks, the sensors are able to collect data and information about new or ongoing attacks like worms, backscatter traffic and scans. The whole of the sensors constitutes a large distributed database thanks to a query engine. An aggregator collects the queries acting as a proxy which forwards requests and responses.

Chapter 3

TTCN-3

3.1 Black-box Testing

Black-box testing [65] is a method that verify the functionalities of a *System Under Test* (SUT) and it is independent from the SUT's internal structure. In order to accomplish to this verification, the tester considers the SUT purpose, the several inputs and the related, expected outputs. Therefore, black-box tests are meant to profile and check the behavior and the function of the SUT. The tester can interact with the SUT by means of one (or more) *Point of Control and Observation* (PCO).
Black-box Testing can be divided into different phases [66]:

- **Test purpose definition**: the final purpose of the test has to be defined in order to formalize the behavior to be tested;

- **Abstract Test Suite (ATS) design**: generation of a collection of test cases derived from an high-level model [67] using a standardized test notation;

- **Executable Test Suite (ETS) implementation**: starting from the ATS, the ETS is built by the so-called *Means of Testing* (MoT) providing the low-level information needed for the test execution;

- **Test execution**: the test is executed against the SUT;

- **Collection of the results**: the test output is one or more verdicts which indicate if the expected and actual responses of the SUT match or not. This comparison is called *pattern matching* and the results can be either passed, failed or inconclusive.

The ATS that will be used is called **TTCN-3**, the ETS will be realize through the MoT called **Titan** which builds the ATS into an executable and provides a testing execution environment where the tests will be executed against the SUT.

3.2 TTCN-3 Introduction

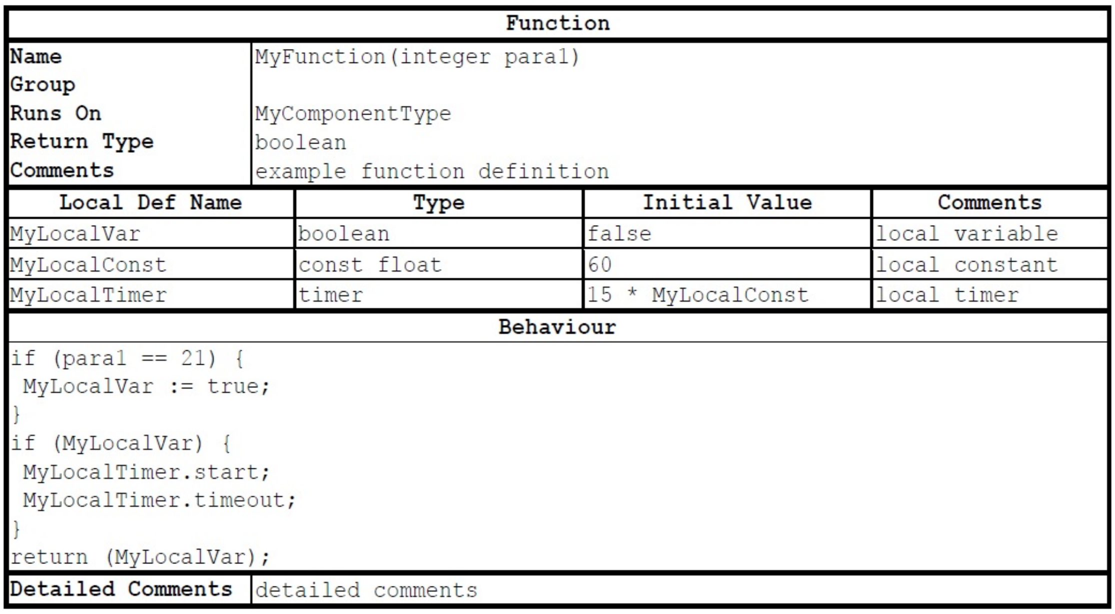
The **Testing and Test Control Notation version 3** (TTCN-3) is a "standardized technology developed and maintained by ETSI and specifically designed for testing and certification" [68]. TTCN-3 and all its components have been standardized in ES 201 873 series [69] which is split into 10 parts that compose the whole standard. These standards have been adopted by the ITU-T in the Z.160 series [70] and continuing through Z.170. It can be applied to several domains such as telecommunication, automotive, medical, power transmission and distribution, financial datawarehouse, avionics and railways. The examples related to the TTCN-3 usage in the black-box testing are numerous, it has been used for defining conformance test suites to SIP [71] and WiMAX [72] standard protocols, the Open Mobile Alliance adopted TTCN-3 for translating test cases into an executable representation and either AUTOSAR and 3GPP promoted the use of TTCN-3 within the automotive and mobile industries, respectively.
TTCN-3 provides all the necessary material for black box testing:

- Abstract Data Types
- Templates
- Event handling
- Timers
- Verdicts
- (A)synchronous communication
- Concurrency

TTCN-3 can be included in the category of the procedural languages since it presents C-like control structures and operators in addition to the aforementioned features. It can be represented in different formats:

- **Core Language**: standard ES 201 873-1, is the textual format and can be edited as text and it is complementary to all the other formats;
- **Tabular Presentation Format (TFT)**: standard ES 201 873-2, is a representation by means of tables (Figure 3.1);
- **Graphical Presentation Format (GFT)**: standard ES 201 873-3, the behavior is defined by means of graphs (Figure 3.2);
- **User defined proprietary formats**

The TTCN-3 Core Language can be integrated with other languages such as ASN.1 types and values, C/C++ functions and constants, IDL, XML and so on.

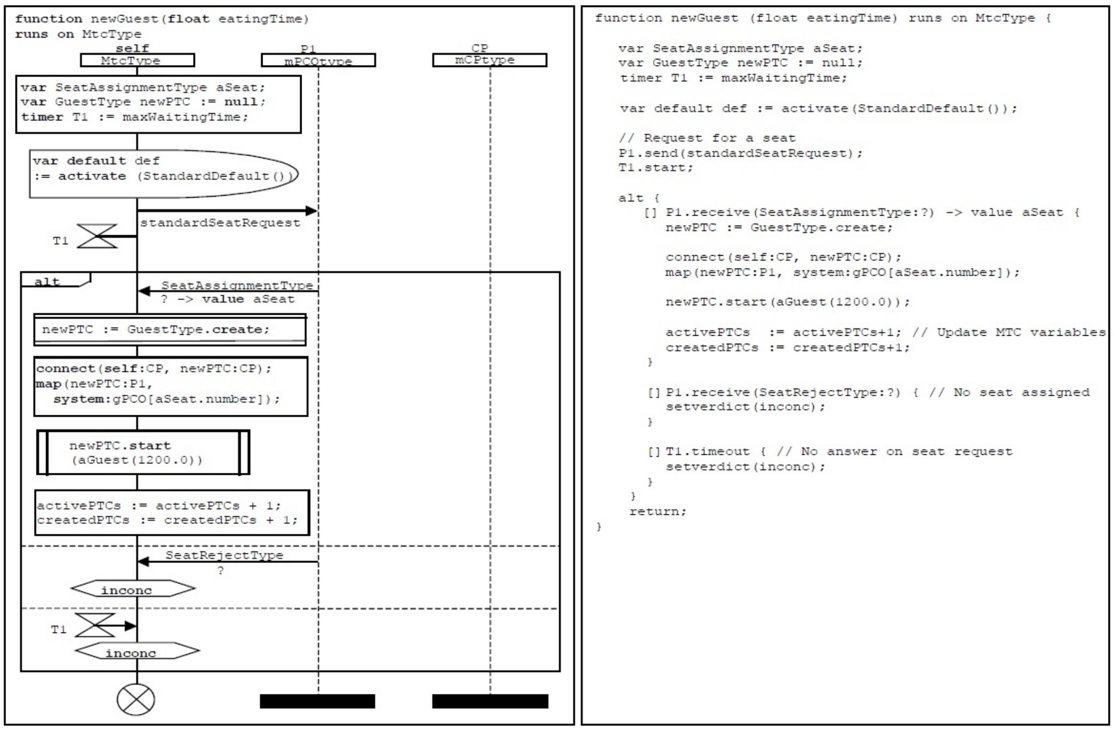


Function			
Name	MyFunction(integer para1)		
Group			
Runs On	MyComponentType		
Return Type	boolean		
Comments	example function definition		
Local Def Name	**Type**	**Initial Value**	**Comments**
MyLocalVar	boolean	false	local variable
MyLocalConst	const float	60	local constant
MyLocalTimer	timer	15 * MyLocalConst	local timer
Behaviour			

if (para1 == 21) {
 MyLocalVar := true;
}
if (MyLocalVar) {
 MyLocalTimer.start;
 MyLocalTimer.timeout;
}
return (MyLocalVar);			

Detailed Comments	detailed comments		

Figure 3.1. Tabular Presentation Format for TTCN-3



function newGuest(float eatingTime)
runs on MtcType
self
MtcType
P1
mPCOtype
CP
mCPtype
var SeatAssignmentType aSeat;
var GuestType newPTC := null;
timer T1 := maxWaitingTime;
var default def
:= activate (StandardDefault())
standardSeatRequest
T1
alt
SeatAssignmentType
? -> value aSeat
newPTC := GuestType.create;
connect(self:CP, newPTC:CP);
map(newPTC:P1,
system:gPCO[aSeat.number]);
newPTC.start
(aGuest(1200.0))
activePTCs := activePTCs + 1;
createdPTCs := createdPTCs + 1;
SeatRejectType
?
inconc
T1
inconc

```
function newGuest (float eatingTime) runs on MtcType {

   var SeatAssignmentType aSeat;
   var GuestType newPTC := null;
   timer T1 := maxWaitingTime;

   var default def := activate(StandardDefault());

   // Request for a seat
   P1.send(standardSeatRequest);
   T1.start;

   alt {
      [] P1.receive(SeatAssignmentType:?) -> value aSeat {
         newPTC := GuestType.create;

         connect(self:CP, newPTC:CP);
         map(newPTC:P1, system:gPCO[aSeat.number]);

         newPTC.start(aGuest(1200.0));

         activePTCs  := activePTCs+1; // Update MTC variables
         createdPTCs := createdPTCs+1;
      }

      [] P1.receive(SeatRejectType:?) { // No seat assigned
         setverdict(inconc);
      }

      [] T1.timeout { // No answer on seat request
         setverdict(inconc);
      }
    }
    return;
}
```

Figure 3.2. Graphical Presentation Format for TTCN-3

3.3 TTCN-3 Core Language Structure

As aforementioned, TTCN-3 Core Language is strongly-typed and presents a C-like syntax and control constructs, with the addition of data types and functions to well represent the parts of a test suite. The latter are data structures and control sections purposely designed to abstract every aspect of a test suite. Some of the most important of them will be introduced to have a better vision and understanding on the implementation.

3.3.1 Module

The Module is the top-level unit [66] and the principle building-blocks of TTCN-3 [73]. Each module is independent from the others and, then, can be compiled as a separate entity. A test suite can contain one or more modules. Each module presents a *definitions part* and can have a *control part*. In the module definitions part there are specified all the data structures and functions that characterize the module such as **constants**, **templates**, **ports**, **components**, **internal functions**, **altsteps**, **test cases**. In order to enable a dynamic behavior of the test suite, it is possible to specify the so-called **module parameters** (belonging to the definitions part as well) through a **Run-Time Configuration File** which can be seen as the arguments of a command line program.
The module control part is optional since a similar (but limited) result can be obtained using the Run-Time Configuration File and defining the sequence of operations and test cases that will be executed. It is the equivalent of the C/C++ function main.

3.3.2 Testcase

A testcase represents the function that must be executed against the SUT in order to check its behavior in relation with the test case purpose. When a testcase is over a final, global verdict is decreed as outcome. TTCN-3 Core Language defines 5 different values of built-in verdicts:

- **pass**: the SUT behaved as expected
- **fail**: the SUT deviated from the correct behavior
- **inconc**: stands for *inconclusive*, is a middle way between passed and failed and it is used when it is not possible to give a correct judgment
- **error**: an error in the test system or devices occured
- **none**: is the initial value for any testcase verdict

The aforementioned values interpretation is the one suggested by [73], but it is actually up to the person who designs the test suite to specify their meanings. During the execution of a test suite it is possible to run several testcases belonging to the same module.

3.3.3 Component

The components are in charge of executing the test cases. In a test case execution there must be at least one component, more than one is optional. If there is only one test component, the execution will be called **Non-Parallel** (or single mode), whereas a **Parallel** execution involves at least two components. For the latter, it is also important to distinguish if the components will lie on the same machine or if they are physically separated; in this case the test is **distributed**. All the components involved in a test suite run in parallel by means of distinct processes forked from the same executable: at the time of their creation, they are told which function or testcase they must execute. When the task is over, the components can wait for a new one or, otherwise, they will be destroyed. This behavior is similar to a thread, since a the components' existence is bound to one (or more) function, but as mentioned before there is one process for each of them. Each component maintains a local verdict and the global verdict will be computed in relation with each of them.
Depending on the role they cover, the components can be classified in the following three categories:

- **Control Component**: executes the control section of a given module, creates the Main Test Component and if the control part is missing executes the testcases specified in the Run-Time Configuration File; it is not an actual component since it does not participate to the execution of a test case but it plays a fundamental role in a global point of view;

- **Main Test Component (MTC)**: is the component that runs the testcase function, then its execution is strongly related to the MTC: it starts when the MTC is created and ends when the MTC is destroyed; therefore in a testcase execution there must be one and only one MTC; in parallel mode it can demand the creation of Parallel Test Components and assign tasks to them;

- **Parallel Test Component (PTC)**: is a component that can be created by any other Test Component (MTC or PTC) according to the purpose of the test suite; when it is started it executes a function and its behavior can be dynamically controlled.

3.3.4 Port

The ports are interfaces that enable the communication between the component they belong to and the SUT (i.e. they are the TTCN-3 solution for PCOs) or between different components. Therefore a communication port is strictly bound to a component and cannot exist in a different scope. The communication can be message-based or procedure-based: in the first case the component sends or receives instances of TTCN-3 or user defined types, whereas the latter allows to send signatures of functions. The ports are full-duplex but some restrictions on the input or output data type can be defined. Incoming data are stored in a FIFO queue different for each port until they are requested by the component owning the port.

3.4 Titan

Titan provides a compilation and execution environment for TTCN-3, developed by Ericsson and submitted to open source via Eclipse Foundation [74]. Titan is, then, a test executor implementing the TTCN-3 Core Language. It actually presents some limitations with respect to the complete TTCN-3 construct set but it supplies some non-standard extensions from the pure TTCN-3 Core Language.
Titan consists of several components which cooperate to provide the full test environment [75].

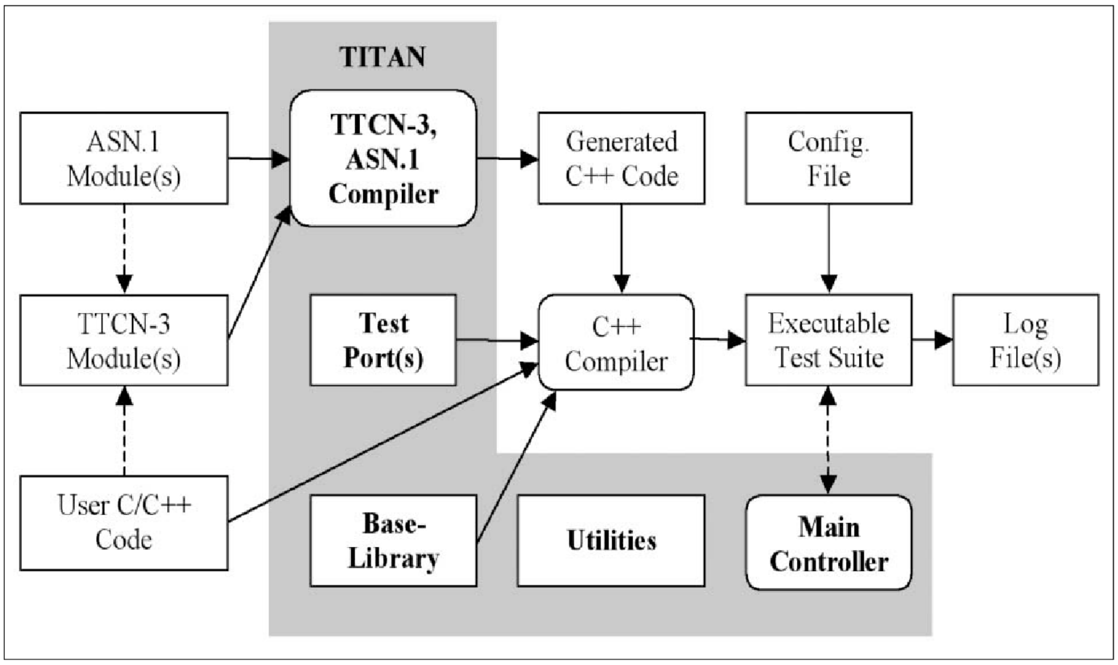


Figure 3.3. Titan components block diagram

3.4.1 TTCN-3 and ASN.1 Compiler

The TTCN-3 and ASN.1 Compiler is the largest and most complex module of Titan Toolset. Titan uses C++ as intermediate language to realize the test environment. The Compiler first parses and analyzes, looking for syntactical and semantic errors, the files written using the TTCN-3 Core Language. Once that has been ascertained that all the TTCN-3 modules are correct, the Compiler generate C++ modules that will next be compiled into executable belonging to the ETS.

3.4.2 Base Library

The Base Library provides the basic information that are independent from the test suite. It is basically made up by static C++ code, already compiled, that is needed to represents the basic features of the TTCN-3 Core Language such as timers, components, ports, verdict handling and so on. Moreover, it contains functions that are used to correctly run the ETS.

3.4.3 Test Ports

In order to allow the communication between the Test Suite and the SUT, is is necessary the presence of Test Ports, which can be seen as a bridge connecting those two worlds. The communication is message-based, they are implemented in C++ and offer a well-defined set of APIs for handling these messages. Their purpose is to convert abstract messages to and from their actual representation. For instance, there is an important presence of Test Port APIs addressing binary encoding and decoding of the most used IP protocols such as Ethernet, TCP/UDP, SSL, DHCP, DNS, HTTP and so on. An exhaustive list can be found in [76]. The following list represents the test ports that are useful for the project scope:

- **IPL4**: is a general purpose test port which allows to access to several transport layer protocols; as regards the network layer, it supports both IPv4 and IPv6, whereas TCP, UDP, SSL and SCTP are the transport layer protocols that can be used in layer 4; the information carried will be the application layer protocol which must be represented as its sequence of bytes;

- **LANL2**: enables to send and receive Ethernet II frames thanks to Packet Socket on Linux; Libpcap is the library used for capturing the packets; in the same way of IPL4 test port, the protocols of layer 3 and beyond must be decoded into byte sequence;

- **PIPE**: the name of this test port could be misleading, it enables to connect with a Unix/Linux shell and execute a given command; the provided functionalities perfectly allows to interact with the shell thanks to the several ways to send input data and the data received in standard input/output or as return code.

The IPL4 and LANL2 test ports exhibit an important limitation for this kind of ports: they are unaware of the protocols of higher layers. In order to overcome this obstacle, Titan provides a set of modules to be imported in the main module of the test suite which enables to encode the sequence of bytes of the protocol header into its abstract representation and vice versa. They are called **Protocol Modules** and include a large set of most used protocol APIs for their handling. In such a way the usage of those test ports is simplified since the higher layers can be easily treated. As for the Test Ports, the complete list of Protocol Modules can be find in [76], the ones that will be useful for our purpose are **IP**, **TCP** and **HTTP**.

3.4.4 Utilities

Titan provides some command line tools to facilitate the several steps concerning the ETS development and execution. Some of most useful are listed here:

- **ttcn3_makefilegen**: takes as input all the files (C++ and TTCN-3) that are necessary for the ETS and generates a Makefile used to compiled the entire project;
- **ttcn3_compiler**: takes as input one or more TTCN-3 files and generates the related C++ code, it is used by *ttcn3_makefilegen*;
- **ttcn3_start**: can be used to execute non-parallel and parallel (but not distributed) test cases;
- **mctr_cli**: it is the utility for launching the Main Controller (3.4.5).

3.4.5 Main Controller

The Main Controller (MC) plays a fundamental role in the the Parallel Test Execution. As explained before, in a Parallel Test Execution there are more than one test components, each of them running in parallel a specific task. The coordination of these components is up to the Main Controller, which is a stand-alone program independent from the test suite program, which orchestrates the whole execution with the following operations:

- creates direct connections to all the components involved in the test;
- handles the creation and termination of the components;
- in distributed tests, if the component assignment is not explicitly done, it distributed the components to the available hosts acting as a load balancer;
- does not take part into the test execution in order to avoid being a bottleneck;
- continuously monitors the test execution, enabling the user to stop the current test at will and start a new one

3.4.6 Parallel Execution Architecture

TTCN-3 Core Language defines the abstract elements which are involved in a parallel test, letting their operation and communication in the hands of the MoT. Titan provides all the concrete elements to make it real. The two main ingredients added to the TTCN-3 pot are the **Main Controller** and the **Host Controller**. The MC has been already introduced in 3.4.5, the HC is an instance of the executable test program and runs in each machine involved in the test. It connects to the MC and when the MC wants to create a new Test Component (MTC or PTC) it demands its creation to the target HC which forks itself and creates a dormant Test Component which will be awakened according to its role into the testcase.
The Figure 3.4 shows the architecture during a parallel execution. Each computer (excluded the one running the MC) runs exactly one HC which maintains a TCP connection

with the MC. This connection is used for the coordination tasks such as component creation, Run-Time Configuration File distribution and so on. Moreover whenever a new Test Component is created, it establishes a new TCP connection with the MC, likewise for coordinating the execution such as function/testcase assignment and verdict collection. The creation of this new TCP connection is due to the absence of any IPC mechanism between the HC and the components instantiated in the same computer.
The MC can be run in two different modes: interactive or batch. In interactive mode the MC is controlled through a Command Line Interface (CLI) which allows to perform several operations such as MTC assignment, test execution handling, Run-Time Configuration file choice and test execution. In batch mode there is no need of a CLI and all the necessary operations are automatically executed. This mode can be set using the Run-Time Configuration file defining a parameter that indicates the number of hosts waited by the MC to start.

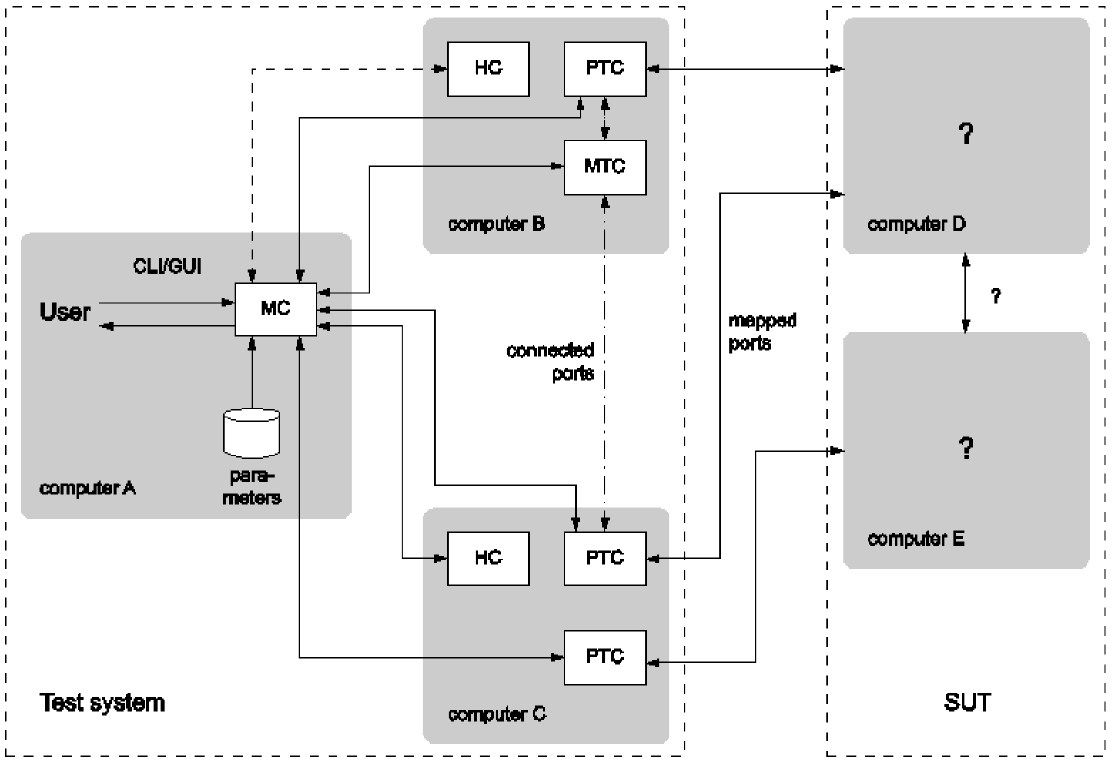


Figure 3.4. Titan Parallel Test Execution

3.4.7 The Run-Time Configuration file

The Run-Time Configuration file describes some aspects of the test suite behavior. It provides some information in the form of parameters that will be used for the whole execution of the test program, even the MC can use its content to adapt its own behavior. These information are stored in a text file, therefore can be modified by the user at will, making the test execution more dynamical since there is no need to re-compile the source code every time that a parameter changes.
It proposes several sections, each of them with a specific scope:

- **Module Parameters**: it contains a list of value assignments to variables which are necessarily defined in the test program; these values can be seen as the parameters used in a command line program and, similarly, they could be optional or mandatory;
- **Logging**: as an additional output, either the MC, the HCs and the Test Components produce log files containing information about the run-time execution; this section enables the modification of their appearance and format;
- **Testport Parameters**: their purpose is very similar to module parameters' purpose; in a Object Oriented Programming point of view, the testports are classes of objects and the testport parameters are their attributes, which must be provided or may be omitted whenever an instance of that object is created; therefore the testport parameters could describe some aspects of their behavior, for instance it can define which port number a TCP server should listen to;
- **Define**: in this section some macro definitions can be defined in order to be successively used in other sections;
- **Include**: within this section it is possible to add the sections defined in other configuration files into the file where the section is declared; the files to be included will be processes exactly once even if it appears several times;
- **Ordered Include**: same as *Include*, but the files are processed sequentially;
- **External Commands**: allows to run command line program in certain points of the execution: when a testcase or control part begins or ends; the commands defined in this section will be run only in the machine hosting the MTC;
- **Execute**: in this section there must be placed the testcase or control parts name that will be launched in the test program execution; in batch mode it is mandatory since defines the testcases run by the test suite;
- **Groups**: here it is possible to define sets of host (by using either their hostname or IP address) which can be used in other sections;
- **Components**: this section is very useful for the distributed tests: it allows to define to which hosts the components (except for the MTC) must be designated; therefore the components will be instantiated to the hosts according to this section and, if missing, the MC will balance the components to the available hosts; a group name

defined in the section *Groups* can be specified as well and in that case one of the hosts in that group will be chosen by the MC;

- **Main Controller**: the behavior of the MC can be defined in here by setting some parameters such as the IP address and TCP port the MC will be listening to for new HCs; it provides a directive to set how many hosts have to be connected to the MC in order to start the execution; therefore this directive indicates if the MC will run in interactive (if omitted) or batch (if defined) mode.

Chapter 4

Implementation

4.1 Architecture

This work focuses on the implementation of the Guardian functions for the missions *Uncover* and *Assess*.
As explained before, those missions, respectively, point to discover if an attack is ongoing and assess the security elements present in the environment in order to find out if they are compromised or they are not working as expected.
What has to be done here is to create a software which provides a first, trivial implementation of the protocol WASSI in order to enable the communication between the Guardians themselves and the master node.
E-WASSI presents a hierarchical architecture where the different hosts are located in different local networks. All of them depend on a **Master Node**, which is responsible to:

- Choose two Guardians from a pool where all the sleeping, pending hosts are waiting for instructions
- Choose a campaign that those Guardians have to execute
- Collect the results and analyze them
- Interpret the results in order to verify if the tested environment is compromised or not.

The Master Node will be always listening for new Guardians to connect and, after an authentication phase, it will maintain a connection with all of them.
Figure 4.1 shows a graphical representation of the architecture in an idle state. The Guardians ignore the presence of the others since the only established connection they have is towards the Master Node. The lines are dashed to represent that the channel is not used but still alive. The Master Node then selects two Guardians that will run the campaign to test one of their network. The Guardian whose network has to be tested will behave as a *client*, whereas the other as a *server*. For the sake of the notation since

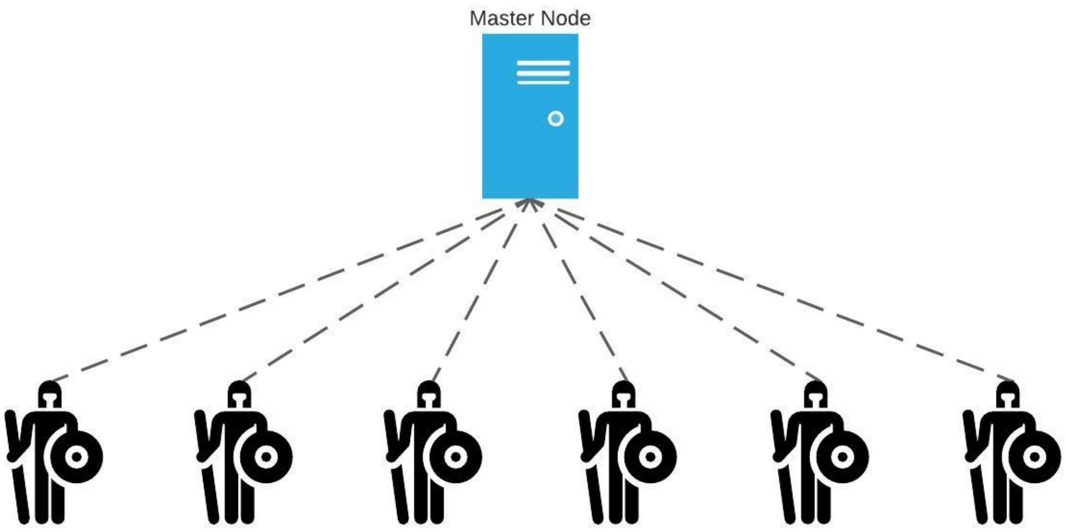


Figure 4.1. E-WASSI Architecture in idle state

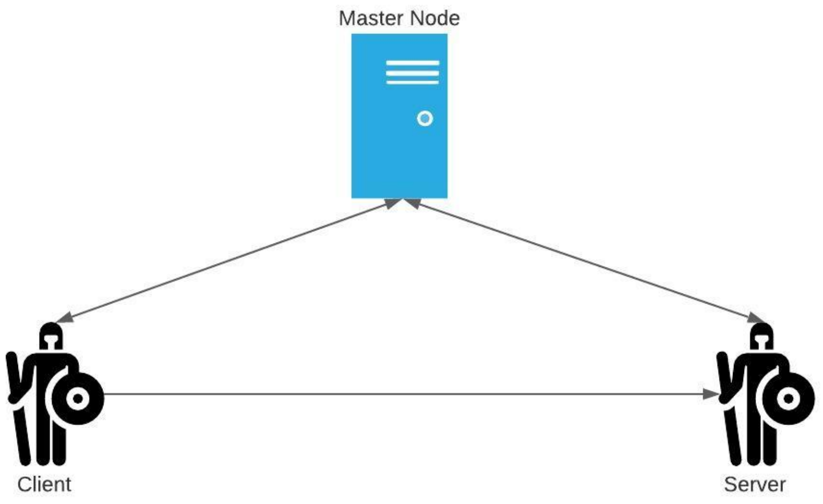


Figure 4.2. E-WASSI Architecture during a test case campaign

the choice of a campaign to its end the two Guardians involved will be called client and server depending on their role. The client takes this name because in most cases it will be the one to start the communication and the server will be one to listen for its connection

(Figure 4.2). Moreover, it will be often (but not always) necessary only to send the request without waiting for a response from the server.
Once that a campaign has been chosen and the Guardians picked from the pool, the Master Node will provide the needed information to the Guardians about their counterparty using the established channel and the various parameters to correctly run the campaign. At this point, the client will start to exchange packets forged with several protocols and headers in order to give an answer to main question carried out by the campaign. During this phase, one or more channels may be established between client and server. When the campaign is over, both client and server send the information collected to the Master Node which will process them. Then the resources allocated for the several test cases will be released and the campaign can be considered over.
The architecture chosen for E-WASSI perfectly fits with the Titan architecture in parallel, distributed mode and moreover TTCN-3 provides all the mechanisms to easily perform test cases and collect their results. In fact there is no need to implement the architecture and all the different roles and connections related to it since they are already natively provided with Titan, making the designer task easier. Moreover TTCN-3 Core language provides advanced mechanisms for templates handling and matching which enable to perform several operations with few lines of code whereas any other programming language would require a much bigger amount. The management of verdicts and log files is extremely handy considering that the verdict collection and the global verdict computation is done by Titan TTCN-3 itself and the log file operations such as creation, opening, editing and closing are not directly handled by the user but through some APIs. The limitations given by the Core language can be bypassed thanks to either the extensions provided by Titan and possibility to compile into the executable C/C++ files that can fulfill all the lacks. The only drawback is the Titan libraries documentations which are located in different git repositories and for someone who just approached to the TTCN-3 coding it could be tricky to jump from one documentation to another, but when you get the hang of it, this won't be an hard task.

4.2 Execution

TTCN-3 and Titan provide the perfect design and execution environments for the E-WASSI purpose. This is true if we consider only the test execution phase, in fact it is necessary to create a pool of Guardians that are waiting for the Master to choose the campaign that will be performed. The whole execution then requires two phases:

- **Pool connection**: in this phase the Master acts as a server, waiting for incoming connection requests from the new Guardians and, once that their identity is confirmed, adding them to the pool;
- **Test execution**: when the Master is ready, it chooses two Guardians from the pool and starts the required test;

What must be noticed is that the two phases can run in parallel since in stable conditions the pool is populated with a sufficient number of Guardians but at the very beginning

the second phase needs the first one to collect at least two guardians that is the lower threshold to correctly run a test.
The proposed implementation takes care of both phases but with some limitations. Two categories of command line programs are proposed: one for the pool management and, on the other side, one for the test execution. The first category includes *Python* programs, one for the Master one for the Guardians, whereas the second one corresponds to the TTCN-3 Titan executable program itself. The latter also includes the MC executable which is provided by Titan and it is not an implementation part of this project. The programs that have been implemented do not aim to give a complete implementation of these two phases but they try to give a starting point for the execution of test campaign, lacking of features that can be successively implemented and that are not strictly necessary for this purpose.

4.3 Pool connection

During this phase the Guardians, by knowing the IP address and the TCP port used by the Master node for accepting incoming connections, start a TCP session with the Master and that connection with the addiction of the Guardian's hostname is used to uniquely identify the Guardian. After that the TCP connection has been established the Master should verify the identity of Guardian in order to not accept other devices that are pretending to be a valid Guardian but it has not been implemented in this work. At this point, the Master picks two Guardians that will run the campaign.
The programs developed for this purpose are tailored for the Master and for the Guardians and, since the architecture is client-server, are called **server_pool.py** and **client_pool.py**, respectively. Both programs use the python module **socket** for the connections and **subprocess** for running the TTCN-3 Titan executable. In this version of the programs, they accept no command line arguments since the interaction is made possible through the standard input. This choice is not obviously the best for running automated tests, in fact it has been simply taken in order to make the debugging and program testing phases easier. Moreover, refactoring the code in such a way that the parameters are passed as program arguments is not an hard task.
The two script will be presented taking into account that both of them participate in either first and second phase and therefore, for the sake of clearness, their involvement in the second phase will be introduced in the Section 4.4.

4.3.1 client_pool.py

In this version, the client_pool.py script is very easy and with few lines of code. Once it has been started, it asks for the IP address of the Master node, whereas the TCP port is hardcoded and then it is constant. After that the TCP three-way handshake has been correctly completed, the Guardian sends one byte encoding the length of its hostname and immediately after as many bytes as the first byte encoded. The hostname corresponds to the value stored in the file with path /proc/sys/kernel/hostname in Linux and its maximum length is defined by the parameter **HOST_NAME_MAX** which, according

to POSIX.1, has a value of 64.
Hereinafter, the Guardians will wait for commands from the Master which, in this version, are either stop the program or prepare for executing a test. The commands are encoded into one-byte long values whose mapping is known by the two parties. The complete

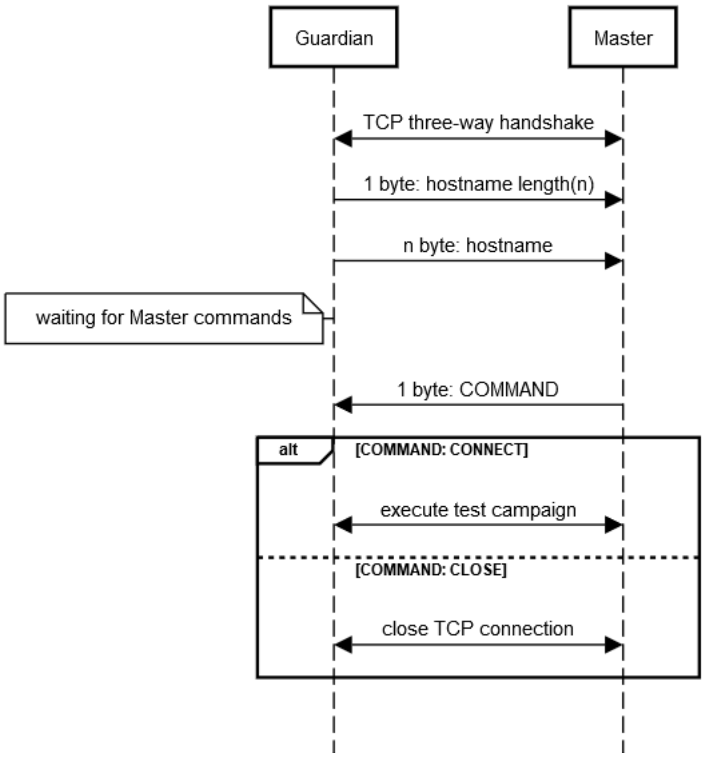


Figure 4.3. Sequence diagram representing the messages exchanged by the Guardian and the Master during the connection phase and the command transmission

sequence of messages exchanged by the two parties during the connection establishment is depicted in Figure 4.3.

4.3.2 server_pool.py

The Master node is started by launching the script_pool.py in the desired host. At the very beginning, the script starts listening to the chosen TCP port and creates a new thread using the module **threading** in charge of handle the commands typed in standard input whereas the main thread keeps accepting new connections. When a Guardian reaches out the Master node, the latter stores it into a dictionary with its hostname as key (consequently the hostname must be unique!) and the information on the connection as value. This dictionary constitutes the data structure representing the pool of Guardian.
In every moment of the execution, thanks to the second thread, the user can type one of the proposed commands which are, in this version, start a test campaign and close

the program. The first command starts the sequence of operations needed to run a test campaign whereas the latter closes the TCP connection iterating this operation for each Guardian laying in the pool.

4.4 Test execution

The test campaign execution is the core of the proposed implementation and it is coordinated by the Master node. It leverages TTCN-3 Titan executable module in parallel mode to run the test cases thanks to the match between either parallel mode and E-WASSI architectures. The test execution involves three different roles and hosts (Figure 4.4):

- **Master node**: this node is the coordination headquarter and therefore all the processes which take place for this purpose will run in this host: server_pool.py that is in charge to handle the connections with the Guardians and actually run the programs of a Titan parallel execution, the Main Controller that represents the Master into the Titan execution environment, and the Main Test Component that coordinates the PTCs during a testcase execution; since for running a MTC an Host Controller is needed, for one test execution four different processes concurring for this purpose will be simultaneously present in the host memory;
- **Client node**: this role is covered by a Guardian which has previously run the script client_pool.py and connected to the Master node; it will act as a PTC in the testcase and then, as for the Master node, an HC will be instantiated;
- **Server node**: it is completely equivalent to the Client node, with the difference of the role covered during the testcase.

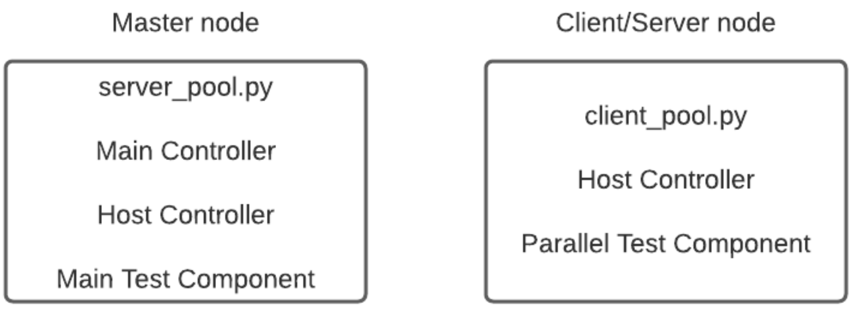


Figure 4.4. Hosts with the different processes operating during a testcase execution

The Guardians that will play the role of the client or server are chosen by the Master among the variety in the pool. In this version it is possible to run only one testcase per campaign but multiple test campaigns can be run one after the other.

In order to correctly start the Titan test suite execution the hosts need to own the files required for their role: the Master node needs the MC executable, the TTCN-3 Titan executable and the Run-Time Configuration File, whereas the Guardians only need the TTCN-3 Titan executable. The Run-Time Configuration file is required only on the Master node because it is up to the MC to distribute it to the HCs in parallel mode. The Master needs the TTCN-3 Titan executable because it runs the MTC and in addition through the executable it is possible to retrieve the names of the testcases available in the test suite. In parallel and batch mode the sequence of steps to be performed is the following:

1. the Master executes **mctr_cli**, the Main Controller, passing as argument the name of Run-Time Configuration file;
2. after checking the correctness of the file, the MC will wait for the minimum number of HCs to be connected as indicated in the Run-Time Configuration file;
3. the Guardians that must participate to the test campaign execute the TTCN-3 Titan executable passing as arguments the IP address and the TCP port of the Master, instantiating a HC;
4. when the required number of HCs is reached, the MC creates the MTC on the first HC that has connected to the Master and distributes the Run-Time Configuration file to all the HCs;
5. the MC demands the MTC to run the testcases belonging to the section EXECUTE of the Run-Time Configuration file;
6. when all the testcases end, the MC destroys the MTC, if still alive, and closes the connections with all the HCs.

What must not be taken for granted is that the MC will always accept the HCs. In fact there are two scenarios where the MC will reject an incoming connection:

- **compiler version incompatibility**: Titan is provided via open source code and then it is possible to build either the compilation and execution environment from it; if a TTCN-3 Titan executable is compiled in a system with different version of the C++ compiler used to build those environments, the MC will reject the HC; it is not even possible to compile a TTCN-3 module into an executable in the same conditions; this is due to the different methods used by different compilers for mapping C++ classes and member function names; according to the last sentence it may be possible to use different versions of the same C++ compiler if that constraint is not violated;
- **ETS version incompatibility**: mctr_cli is a stand-alone program, then it does not need to know the TTCN-3 Titan executable which will be used; considering that an HC connects to MC using the ETS, the MC will set as accepted/correct version of the ETS the version of the ETS used by the firstly connected HC; moreover since a TTCN-3 module can include and import other modules, for each module a version is set with the same logic of the module of the TTCN-3 Titan executable; in the case

that one or more module version differ with respect of the firstly connected HC, the MC will reject the other HC which will not satisfy that condition.

4.4.1 Master node

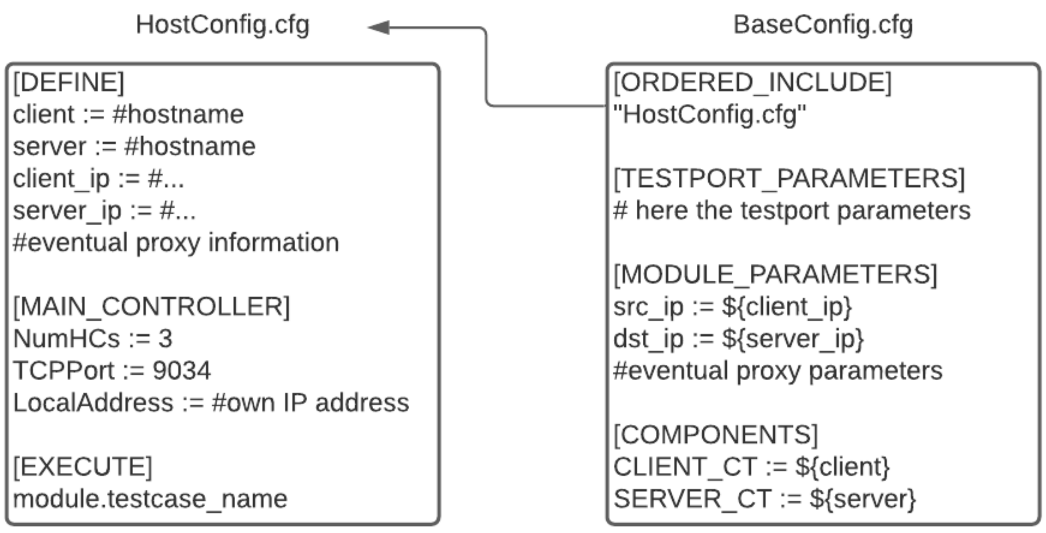
When the command for starting the test cases has been typed in the interface provided by server_pool.py, the script firstly lets the user choice a testcase among the ones available in the ETS. From the testcase name it is possible to build the name of the Run-Time Configuration file corresponding to that testcase (see 4.4.3). Subsequently, it asks the user to choice two Guardians among the ones laying in the pool that will play the role of client and server. The last step is to choose if the testcase has to run in presence of a proxy server. It it is the case, the program lets the user type the IP address of the proxy, the port used for the forwarding operation and if the proxy will use the domain name or the IP address of the server in the Host header. If the user chooses the domain name, it has to be typed as last operation in standart input. Once that the choice is taken, the Run-Time Configuration file will be modified using the python module **configparser** in order to contain all the information about the hosts necessary for the testcase. These modifications are testcase-independent.
At this point everything is ready to start a testcase. The script, thanks to the module **subprocess**, initially runs the MC passing the correct file name for the chosen testcase, then when the MC is ready to accept HCs the script creates the HC for the Master node and connects it to the MC. This synchronization method is necessary in order to create the MTC on Masternode itself. Afterwards, the Master tells the designated Guardians through the established TCP connection that they can now starts an HC and connect it to the MC.
The task of the script is now finished and it must wait for the MC to be terminated.

4.4.2 Guardians

The Guardians during the preparation phase will just be waiting for new commands from the Master since they are not needed. When they receive the command to start a test, what they do is simply starting the HC process by running the ETS and passing the IP address and TCP port used by the MC, and then waiting for the latter to be terminated. When the testcase is over, they will be back to the pool and be available for a new execution.

4.4.3 Run-Time Configuration file structure

As aforementioned, the Run-Time Configuration file provides important information, often mandatory, for the correct execution of the ETS. Since most of the parameters are often the same by changing the testcase and only the parameters specific for the testcase can be different, the structure of those file defines a common part shared by all the testcases and a part that is typical for different testcases. The common part is constituted by two different files: **HostConfig.cfg** and **BaseConfig.cfg** (Figure 4.5).



```
[DEFINE]
client := #hostname
server := #hostname
client_ip := #...
server_ip := #...
#eventual proxy information

[MAIN_CONTROLLER]
NumHCs := 3
TCPPort := 9034
LocalAddress := #own IP address

[EXECUTE]
module.testcase_name
```

```
[ORDERED_INCLUDE]
"HostConfig.cfg"

[TESTPORT_PARAMETERS]
# here the testport parameters

[MODULE_PARAMETERS]
src_ip := ${client_ip}
dst_ip := ${server_ip}
#eventual proxy parameters

[COMPONENTS]
CLIENT_CT := ${client}
SERVER_CT := ${server}
```

Figure 4.5. Common Run-Time Configuration files structure

HostConfig.cfg is the only file that is modified by the Master script. First of all it provides the information related to the hosts as the file name suggests: client IP address and hostname, server IP address and hostname and eventually the proxy information such as IP address, TCP port, host to reach and so on. This information is deployed to all the Run-Time Configuration files which will include HostConfig.cfg since they are provided in the Define section. HostConfig.cfg defines the MC behavior through the section Main Controller: it sets the batch mode by defining the parameter which says how many HCs must be connected to the MC to start the execution and assigning to it the value 3 (1 MTC, 1 PTC client, 1 PTC server); it also sets the MC TCP port to 9034 (value recommended by [77]) and the IP address; at last it inserts the testcase name chosen by the user into the section Execute.
BaseConfig.cfg contains all the basic parameters that are common between all the testcases and therefore is included into each configuration file belonging to a specific testcase. It includes HostConfig.cfg and, thanks to that, assigns correctly the components to the hosts using the hostname. Moreover it sets the testport and module parameters shared among all the testcases.
The configuration files specific for each testcases can vary depending on the testcase they are related to. Their names can be obtained by truncating the ending word *Test* from the testcase name and then concatenating *Config.cfg*. For instance, if the testcase name is *testcase_nameTest*, then the corresponding configuration file name is *testcase_nameConfig.cfg*, in such a way it is easily possible to associate the file name to the correct testcase. These files should be different for each testcase but they are pretty much

the same for each testcase since they do not present big differences in terms of initial settings. Then the specific Run-Time Configuration files implemented in this version are only two: one for the testcases where a normal TCP connection is used and one with the information for SSL/TLS. The former defines the packet filters used by the LANL2 testports and the identification field of the outgoing IP packets whereas the former includes the filepaths of the SSL certificate chain, the private key and the trusted CA list which are mandatory for the testport IPL4 when used on the server for listening for incoming SSL/TLS connections.

4.5 Testcases implementation

Once that the structure surrounding the ETS execution has been analyzed, it's time to introduce the several testcases that have been designed and implemented with the final missions of Uncover and Assess. The testcases are the actual representation of the WASSI protocol talked by the client and the server. Each testcase aims to analyze different aspects of the sent and received packets at any layer in order to verify that one or more fields under investigation show the expected value rather than a wrong one. The testcase analysis will describe firstly its task and goal and then it will focus on the actual implementation in TTCN-3. Some parts shared among all the testcases and related to the TTCN-3 module structure will be treated separately for the sake of clearness. Moreover, the WASSI protocol always uses HTTP messages in the testcases even if for some of them it can be avoided. This choice is due to either make sure that the packets look like actual messages and in the case a proxy is used. For each testcase the final output will be represented by either a verdict and a series of log file. The verdict is one of the aforementioned values, which must be interpreted in the context of the testcase it belongs to. A better explanation of the meaning of the verdicts is given by the log files which are one for each process (MC, HCs and components) involved in the testcase. The log files are edited by the processes themselves, then they will be stored in the host running the related process. These files contain also other information regarding the run-time execution and operations in order to have a clearer perspective on what happened during the testcase. In conclusion, verdicts and log files are complementary and then must be interpreted as an unique entity. For instance, a global verdict set to fail does not necessarily mean that testcase did not go as expected but it is just a convention used to express that the testcase converged on that result rather than another one.

4.5.1 Components

As previously described, all the testcases will involve three hosts to whom are associate three different components, one for each of them. In the current implementation, more than three component types are defined but only three of them will be instantiated and a processes corresponding to that component will be created:

- **MTC_CT**: it is a dumb component type representing the MTC; it only contains as internal variables the other two components;

- **CLIENT_CT**: this component represents the client and will the one which will start the appropriate communication with the server; it owns several testports which will be introduced later; it inherits from another component called HOST_CT used as base component with the server for the common attributes;
- **SERVER_CT**: it is totally equivalent and presents almost no differences with CLIENT_CT which in most cases are the internal variable names; the choice of creating two different component types is due to firstly distinguish them in the component assignment and for future implementation where this distinction will be mandatory.

There is actually another component type that is involved in the testcases but does not need to be instantiated in any process: it is called **SYSTEM_CT** and it is an abstract representation of the SUT. It owns the same testports of all other components which are used as sockets where the correct plugs, that are the component testports, must be inserted.

4.5.2 Testports

The testports cover a fundamental role in the ETS because they are the access and observation point towards the SUT. In our scenarios the testports are used to connect the client and the server through the SUT that is the whole network between them. Several testport types are owned by the components, each of them have a different purpose:

- **PIPEasp_PT**: this testport is an exception to what has just previously said, in fact it is not used to communicate with the SUT but to execute shell commands and scripts; its usefulness will be better explained later;
- **IPL4asp_PT**: as explained before, through this testport the component can send or receive layer 4 packets, accessing only to the information of the application layer and some information of the lower layers such as source/destination IP address/port;
- **LANL2asp_PT**: as for the IPL4 testport, this testport allows to send Ethernet frames; in this case it is possible to modify all the fields of the Ethernet header; it is useful when the IPL4 testport is not enough: for instance if a field of the IP header must be modified in order to assume a "unnatural" value, the LANL2 testport can help in that purpose; it is easily to convey that this testport could be inconvenient because every higher layer must be forged byte by byte, fortunately Titan helps by providing functions that make it easier but it still preferable to use the IPL4 testport whenever it is possible.

All of these testports are used either for sending and receiving information. The sending operation is non-blocking, it means that the execution of the program is not blocked by that, whereas the receiving operation is blocking. In order to avoid endless waiting which lead to starvation/deadlock of one or more components, for each receiving operation a timer is started and if the time expires the execution proceeds taking into account this event.

4.5.3 Useful functions

Some operations that are needed for preparing the components to correctly execute the testcases are often repeated for each testcase. Here they will introduced in order to avoid repetitions later:

- **createIPpacket**: it is used to create the byte array containing either the IP header and its payload; all the fields belonging to the IP header are passed to the function with usual types such as integer, strings an so on and the function, leveraging the protocol module IP, producing as output the binary encoded packet;
- **createTCPsegment**: it is the corresponding of createIPpacket for TCP segments, what changes here is the parameters passed to the function that are fields of the TCP header and its payload; the output of this function is usually passed as payload to createIPpacket; it uses for this purpose the protocol module TCP;
- **createHTTPrequest**: it is used to create HTTP requests as suggested by the name; the input parameters are the headers of an HTTP request and the output is the binary encoded message, using the protocol module HTTP;
- **createHTTPresponse**: since the HTTP request and response are slightly different, the encoding of these two types of messages has been split off into two distinct functions; moreover the API provided by the protocol module HTTP is different for request and response, then it was more convenient;
- **getInterfaceName**: can be called by either CLIENT_CT and SERVER_CT and, by leveraging the testport PIPE, it returns a string containing the name of the interface currently connected to the Internet; it is used whenever the testport LANL2 is used because it needs to know which interface must be used to send the frames;
- **getGatewayMACAddress**: by using the testport PIPE it returns a string with the MAC address of the gateway as a string; it does not send any ARP query but uses the routing and ARP tables of the system to retrieve it; in the current version if the destination host is in the same LAN of the source host, the MAC address retrieved will be the one associated to the destination host; this function is used in the same scenario of the previous function since the MAC address will be used as field of the Ethernet header by the LANL2asp_PT testport;
- **getHopNumber**: this function is used only for the testcase SmallTTLTest (4.5.7) and returns an integer representing the number of hops between the source and destination hosts; this value is computed through a script called by the PIPE testport which leverages the output of the command *traceroute* to whom the destination host IP address is passed in order to compute the number of hops;
- **perform3WayHandshake**: when the testport LANL2asp_PT is used for the client-server communication, since it only sends frames it does not perform the TCP three-way handshake; this could lead some packets to be dropped by the firewalls operating between the two hosts or by the server itself since the packet does not belong to any

established connection; this function can be run only by CLIENT_CT and properly sends and receives the segments of the three-way handshake; it firstly uses a IPL4asp_PT testport to perform the handshake since the testpost creates a TCP socket with the SERVER_CT which is using the same type of testport for listening for new connections; during the handshake the LANL2_PT testport sniffs the traffic filtering only the ACK segment sent by the client in order to obtain the correct sequence and acknowledgement numbers that will be used for sending frames, after the handshake, through the LANL2asp_PT itself; the server is properly using the IPL4asp_PT and the kernel of the system will reply with ACK segments which, once they will have reached the client, will trigger a response from the client's kernel that will send RST segments since they are "acking" segments that the kernel does not know that have been sent; in order to avoid this behavior, when the kernel must send the RST segments, they are pruned and dropped thanks to an iptables rule added by this function on the client which prevents outgoing RST segment having IP address and TCP destination port of the server to leave the host; this rule is set by the testport PIPE;

- **closeTCPConnection**: this function is the counterpart of the previous function and performs the opposite operations; it firstly closes the client-side TCP connection with the server using the testport IPL4asp_PT and then deletes the iptables rule previously created;
- **startMTC**: this function is called by MTC_CT at the very beginning of each testcase and creates the components CLIENT_CT and SERVER_CT, which at this point will be assigned to the correct hosts as indicated in the Run-Time Configuration file, and initialize their testports; considering that this operations are executed in the Master node and the other hosts are located in other local networks, the HCs of either client and server will be reached out by the MC, in its turn contacted by the MTC requiring those remote tasks;
- **stopMTC**: it performs the opposite operations of startMTC, destroying testports and components; as for the previous function, it is called for each testcase but obviously just before the end of the testcase.

4.5.4 FragmentationTest

This testcase aims to reveal network devices that reassembly fragmented IP packets. The client sends to the server an HTTP message with a small size, smaller than the usual MTU value of 1500 bytes, split into two IP fragments. When the first packet reaches the server, the latter checks if its More Fragments bit is set: if it is the server waits for the second fragment for the sake of completeness. If the packet is not fragmented then it is possible to deduce that something has reassembled the packet. Since the packets could arrive out-of-order, if the server will receive the second fragment for first, it will obviously wait for the first fragment (Figure 4.6).

The SERVER_CT component starts for first and waits for client to be connected. Since the packets that must be forged have to be modified at IP layer, the CLIENT_CT

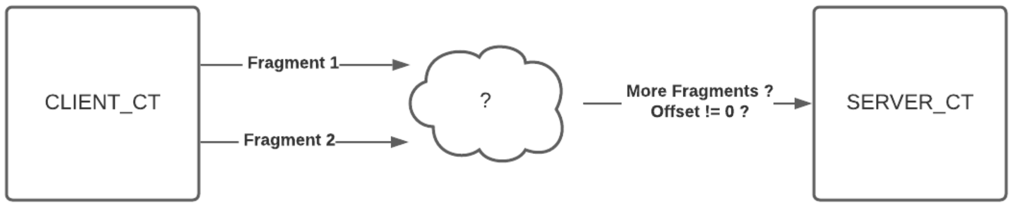


Figure 4.6. Fragmentation testcase

component will use the LANL2asp_PT testport for this purpose. It creates a dumb HTTP request, the TCP header and then splits the latter, to whom the HTTP request has been appended as payload, into two fragments paying attention that the size of the first is a multiple of 8: the first fragment will show an IP header whose More Fragment and Offset fields are 1 and 0 respectively, whereas the second fragment's IP header has the field More Fragments set to 0 and Offset with as value the size of the first fragment divided by 8. The SERVER_CT component will leverage its own LANL2asp_PT testport to sniff the traffic coming from the other component and access to the fields of the IP header. The final verdict is established as follows:

- **pass**: only if the SERVER_CT component correctly receives the two fragments;
- **fail**: if the SERVER_CT component receives no packets, one single non-fragmented packet or only one of the two fragments;
- **inconclusive**: if any timer in the client or the server expires;
- **error**: all the other cases;

4.5.5 SequenceNumberTest

In this testcase the client will send a dumb HTTP request to the server and both of them will store the sequence number of the TCP header underlying the HTTP request. The two values will be sent back to the Master which will compare them to find out if the TCP header field has been modified (Figure 4.7).

Likewise the previous testcase, the CLIENT_CT and SERVER_CT components will use the LANL2asp_PT instead of IPL4asp_PT testpost because the latter does not allow to modify or retrieve the sequence number of the sent or received segments. When the HTTP request arrives to the SERVER_CT component, it will extract its sequence number and both the CLIENT_CT and SERVER_CT components will send back to the MTC_CT the values they collected via the MC using the TCP connections established between the components and the MC at their creation time (this is possible thanks to the Titan language extension *done* [78]). The MTC owning both values can compare them and decide the final verdict.

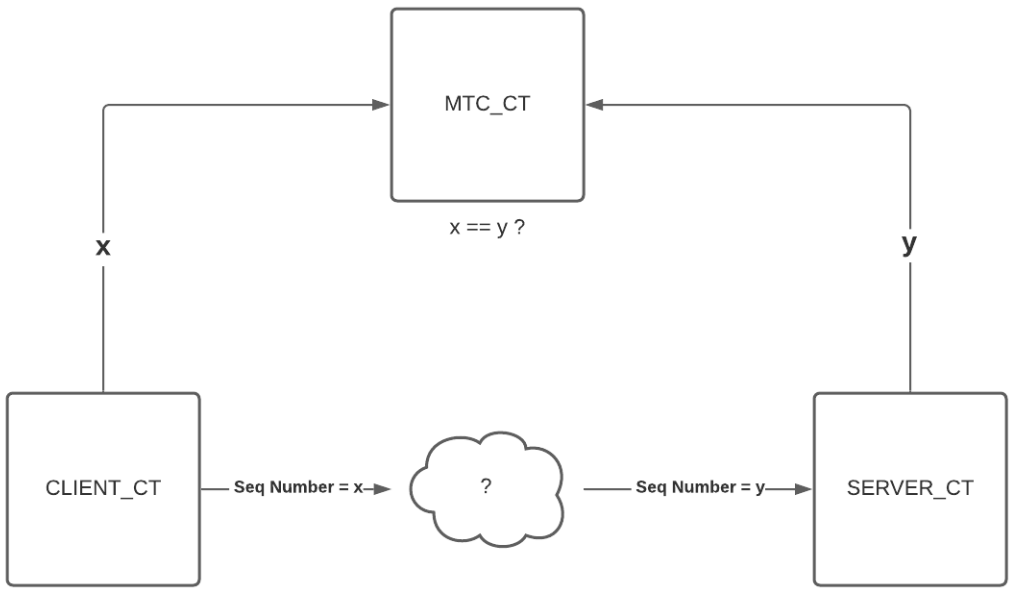


Figure 4.7. Sequence number testcase

- **pass**: only if the sequence number sent by the client and received by the server match;
- **fail**: only if the two sequence number do not match;
- **inconclusive**: if any timer in the client or the server expires;
- **error**: all the other cases;

4.5.6 TLSKeysTest

The client and the server start a TLS session e when the handshake is completed both of them, using an exporter according to RFC5705 [79], retrieve a pseudorandom bit string generated from the master secret which is obviously always the same if the shared key does not change. Those strings are sent back to the Master that will check if they are the same or if not, sign of a possible ongoing Man In The Middle attack (Figure 4.8).

For this testcase the IPL4asp_PT testport is enough because it is able to establish a TLS session between CLIENT_CT and SERVER_CT components. It firstly starts a TCP connection and on top of that it starts TLS. When the handshake is completed, the components use an exporter provided by the testport itself through an API to retrieve the bit strings. The exporter needs some parameters to be passed to correctly export the information. These parameters are shared between the two components and are a label which is the hexadecimal number 0xdeadbeef, a context which is the string

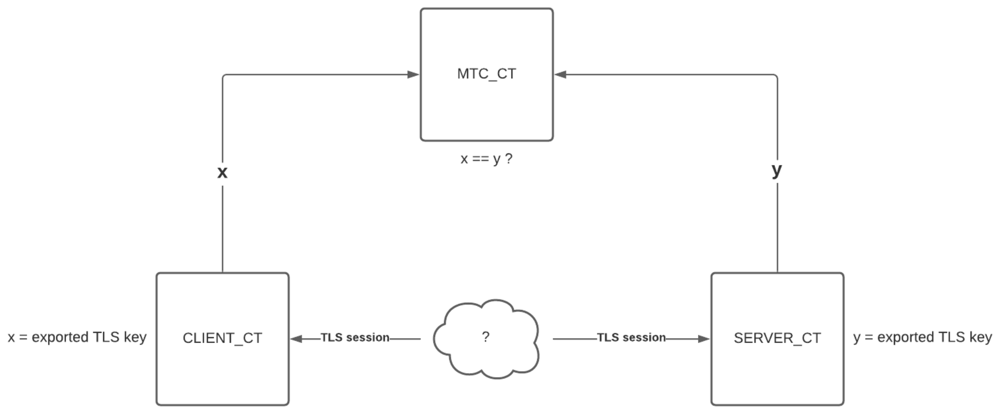


Figure 4.8. TLS keys testcase

"ewassi_tls_exporter" and the length of the output that is 1024 in this version. The first two parameters can be modified thanks to module parameters of the Run-Time Configuration file, whereas the latter is hardcoded so it is immutable. In the case that a proxy is used the operations performed by the CLIENT_CT component are slightly different, whereas the SERVER_CT component does not change (Figure 4.9). First of all the client starts a TCP connection with the proxy and through that connection sends a HTTP request having method CONNECT, the domain name or IP address of the server followed by ":443" as requested host and the server domain name or IP address as Host header. The proxy will start a TCP connection with the server and when it is correctly completed sends a HTTP response to the CONNECT with status code 200 if it was able to connect to the server. At this point the CLIENT_CT component starts the TLS session with the proxy that, if its working as supposed, will forward the messages of the TLS handshake to the server, creating an end-to-end session between the client and the server passing by the proxy itself. When the exporter has accomplished its task, the two bit strings are pushed to the MTC_CT component in the same way of the previous testcase that will check if they match and decree the final verdict:

- **pass**: only if the exported keys match;
- **fail**: only if the exported keys do not match;
- **inconclusive**: if any timer in the client or the server expires;
- **error**: all the other cases;

4.5.7 SmallTTLTest

The client sends a normal HTTP request to the server with a very low Time To Live (TTL) value in the IP header. If the server receives that message, it means that someone

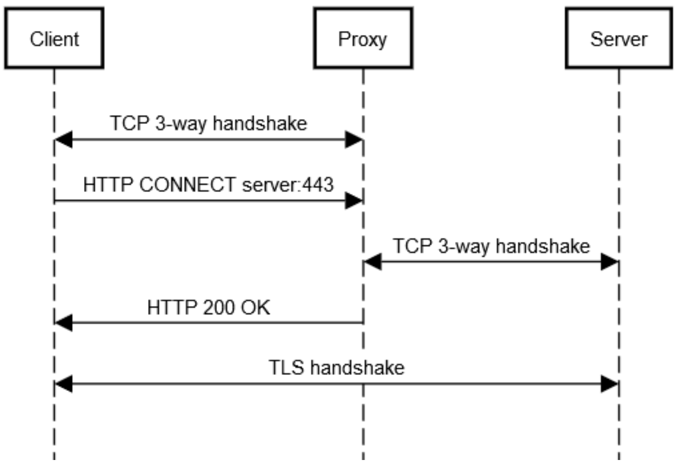


Figure 4.9. Sequence diagram for HTTP CONNECT

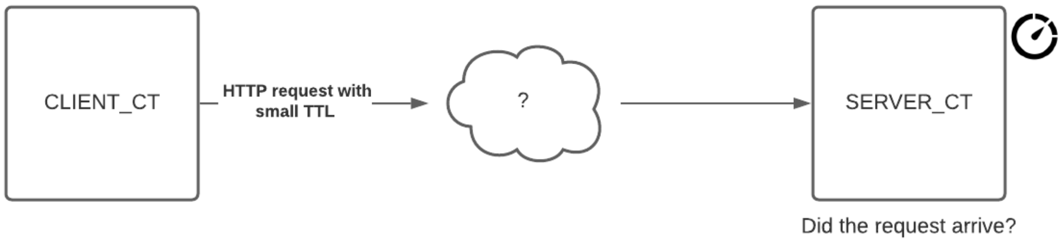


Figure 4.10. Small Time To Live testcase

modified the packet and reset the TTL. If the server does not receive any packet after a predefined period of time, that means that the packet expired and then there were no modification on that field (Figure 4.10).

The CLIENT_CT component computes the number of hops required by a packet leaving the client to reach the server thanks to the function *getHopNumber*. When the command successfully ends the CLIENT_CT component set the returned value decreased by 1 as TTL of the IP layer of the forged HTTP request and then sends it out through the LANL2_PT testport to the server. On its side, the SERVER_CT component is listening for incoming messages from the client and it stops listening after an hardcoded value of seconds (30 in this version). The global verdict is set as follows:

- **pass**: only if the timer regulating the waiting time for incoming packets on the SERVER_CT component expires;
- **fail**: if the server receives the HTTP request;

- **inconclusive**: if any other timer in the client or the server expires;
- **error**: all the other cases;

In the case where a proxy is used, if the script computes the number of hops between the client and the server, this data could be meaningless since either the route taken by the packets can vary depending on the physical location of the proxy and the proxy itself will probably generate an new IP header with a new TTL value. It has been decided that in this case the computed number of hops is related to the pair client-proxy rather than client-server. In such a way the testcase tries out only "half" of the network between the two hosts, up to the proxy. In order to have a complete perspective, the testcase can be run once again but by swapping the client and server roles.

4.5.8 MD5Test

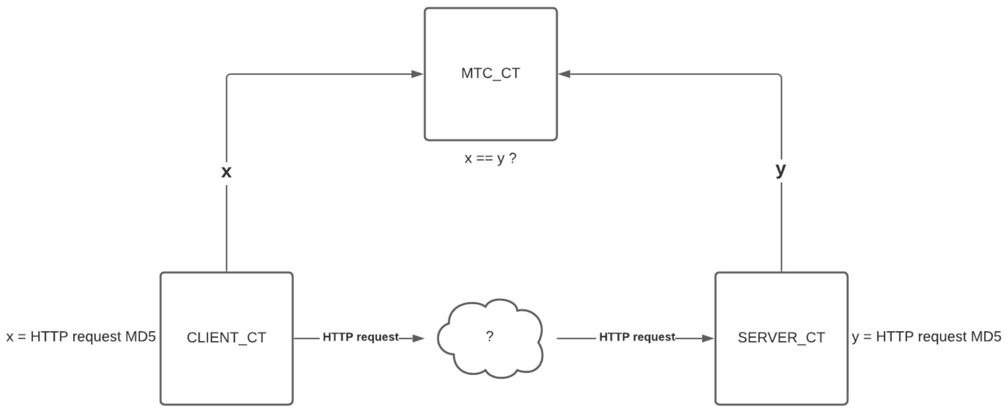


Figure 4.11. HTTP request MD5 testcase

The client sends a normal HTTP request to the server, calculating the MD5 hash of the application layer bytes. The server on its side does the same with the received HTTP request. Both of them send the computed hash to the Master which if any modification has been introduced in HTTP request (Figure 4.11).

The CLIENT_CT component uses the IPL4asp_PT testport to send the HTTP request since there is no need to modify the IP or TCP header. When it forges the HTTP requests thanks to the function f_calculateMD5_oct provided by the Titan library **TCCUsefulFunctions** it can compute its MD5 hash. The SERVER_CT component will use the same function and both will return the values to the MTC_CT component which will check if they coincide. The global verdict is set as follows:

- **pass**: only if the hashes match;
- **fail**: only if the hashes do not match;

- **inconclusive**: if any timer in the client or the server expires;
- **error**: all the other cases.

4.5.9 XForwardedForTest

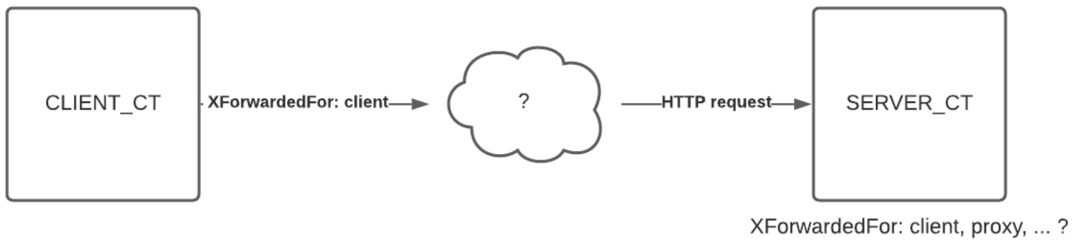


Figure 4.12. XForwardedFor testcase

The client sends an HTTP request to the server containing the header **X-Forwaded-For** with as value its own IP address. This header is used in the presence of proxy servers which every time the forward a packet they could append their IP addresses to the ones already existing. Thanks to this mechanism, the receiver of the request can learn the actual source of the requests and the IP addresses of the servers that have forwarded the message. When the server receives the message it checks if that field contains only the IP address of the client or if others have been added (Figure 4.12).

The CLIENT_CT component sends the HTTP request with the X-Forwaded-For header through the IPL4asp_PT testport which, once that the SERVER_CT component has received it, it will check if a proxy added its own IP address or not without sending that value to the server. That is possible because the server knows the client IP address since it is defined as module parameter and provided to all the components. The final verdict will be set according to the following rules:

- **pass**: only if the X-Forwarded-For header contains only the IP address of the client;
- **fail**: only if the X-Forwarded-For header contains the IP address of the client and other IP addresses;
- **inconclusive**: if any timer in the client or the server expires;
- **error**: all the other cases.

Chapter 5

Evaluation

The Titan TTCN-3 compilation and execution environment has been built from the source code [76] on a kali 2020.3 machine with kernel Linux 5.8.0-3-amd64 x86_64 and using gcc as compiler with version 10.2.0. As aforementioned the gcc version is fundamental for the ATS compilation and ETS execution. The Titan version is 7.1.pl1.
The Makefile used to compile the ATS into the ETS is generated with the command:

```
$ makefilegen -f -e ewassi *.ttcn *.cc *.hh *.h *.c *.l *.y
```

where *ewassi* is the name of the future executable and it is followed by the filenames of the TTCN-3 modules, C/C++ source code and all the other files that are required by the modules. In this case every file with those extensions is included since in the directory all those files are required. *makefilegen* is a link for *ttcn3_makefilegen* presented in 3.4.4. The output is a Makefile including almost all the information necessary for compiling the ATS; in fact some libraries used by some Titan Testports do not figure since the command *makefilegen* does not know which libraries are required. The installation guide of the Testports can be found on the docs of each of them. In the Makefile generated for the implemented ETS what has been added is the openssl directory path, required for the SSL/TLS sessions, and a series of dynamic libraries such as pcap, util and ssl. The C/C++ files are the actual implementation of the testports then they define the low-level structure and functions run by themselves. Regarding the testport LANL2asp_PT, in run-time it presented a problem when the packets had to be delivered to the TTCN-3 function. Since that testport does not create any socket, the packet are captured by using the **pcap** library. The function in charge of opening an interface in capture mode is *pcap_open_live* whose arguments include one called packet buffer timeout which allows to buffer the sniffed packets within a time interval expressed in milliseconds and deliver all the packets captured in that period at the same time. On the provided testport that value was set to zero, then the testport was operating in immediate mode, that is the packets should be delivered as soon as they arrive. For some reason the function did not returned immediately the packets, on the contrary it waited for an apparently unpredictable amount of time before delivering the packets. This issue has been fixed by replacing the value zero with 100 milliseconds in the testport C++ file LANL2asp_PT.cc. The command *make* then creates the ETS with the name used in the *makefilegen*.

The ETS has been executed in the GNS3 environment depicted in Figure 5.1 which

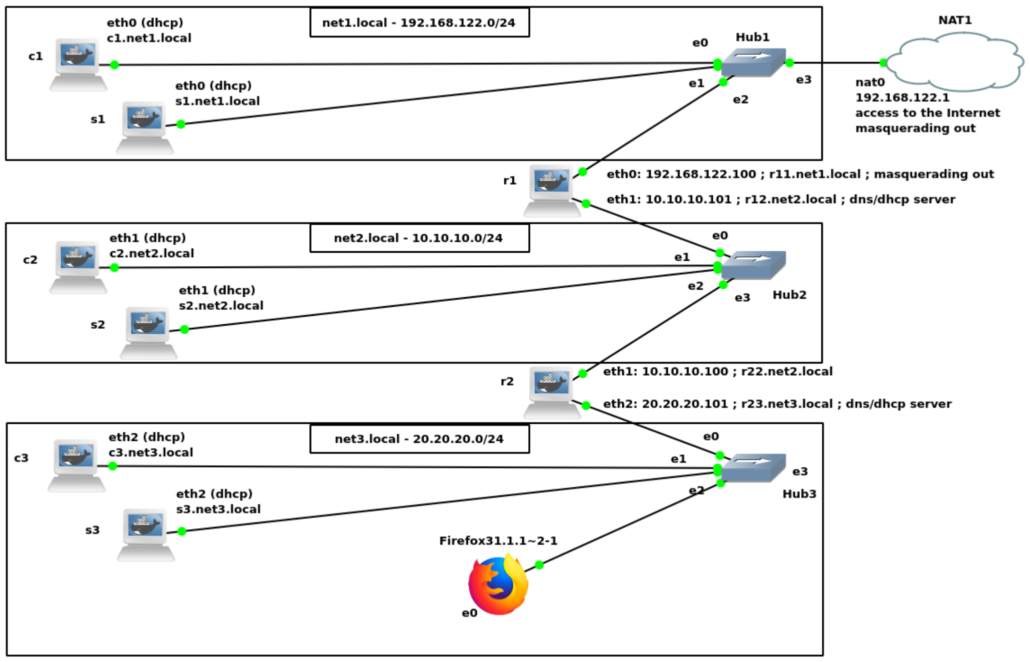

net1.local - 192.168.122.0/24
eth0 (dhcp)
c1.net1.local
c1
eth0 (dhcp)
s1.net1.local
s1
e0
Hub1
e1
e2
e3
NAT1
nat0
192.168.122.1
access to the Internet
masquerading out
r1
eth0: 192.168.122.100 ; r11.net1.local ; masquerading out
eth1: 10.10.10.101 ; r12.net2.local ; dns/dhcp server
net2.local - 10.10.10.0/24
eth1 (dhcp)
c2.net2.local
c2
eth1 (dhcp)
s2.net2.local
s2
e0
e1
e2
e3
Hub2
r2
eth1: 10.10.10.100 ; r22.net2.local
eth2: 20.20.20.101 ; r23.net3.local ; dns/dhcp server
net3.local - 20.20.20.0/24
eth2 (dhcp)
c3.net3.local
c3
eth2 (dhcp)
s3.net3.local
s3
e0
e1
e2
e3
Hub3
Firefox31.1.1~2-1
e0

Figure 5.1. GNS3 topology

presents the following settings:

- All the end devices and routers (except for the Firefox host) runs a Docker container with image **ninovitale/ewassi:1.1**, which is an Alpine Linux based container with the autoconfiguration for clients, servers and routers based on their hostname and with all the libraries required by the Titan execution environment and the developed ETS;

- The hosts used during the ETS execution are only located in **net2.local** and **net3.local** because the router r1 has an iptables rule in the POSTROUTING chain for masquerading the packets going out the interface eth0 and that rule can cause problems for the packet filters and component assignments during the testcase execution; that rule cannot be removed because it gives Internet connectivity to the hosts in the those two networks;

- the proxy deamon used for the proxy server is **tinyproxy** [80] because it is small and supports HTTP, SSL/TLS and the CONNECT method.

All the testcases previously introduced have been run with the following component assignments:

- **s3**: runs *server_pool.py* and then it will be playing also the roles of MC and MTC;

- **c3**: runs *client_pool.py* and it is designed to be the component CLIENT_CT;
- **c2**: runs *client_pool.py* and it is designed to be the component SERVER_CT;
- **s2**: runs *tinyproxy* with a simple configuration file to enable HTTP, SSL/TLS and CONNECT forwarding; the only non-standard modification made to the HTTP request is to append "tinyproxy1" to the header Via which, if does not exist, is created.

Ther results of the testcases are the following:

- **FragmentationTest**: the testcase always fails because the packet arrives to the server reassembled and not fragmented, even if a proxy is not used; this behavior is bizarre since there are no devices which need to access information related to the TCP layer or above; the main supposition is that the router r2 reassembles the packet only because it is running iptables even if no rules are defined;
- **SequenceNumberTest**: the result depends on the presence of the proxy server, in fact it always failed in the case it is used because the TCP connections between proxy and client and proxy and server are different, whereas in the case of direct connection between client and server the verdict is pass; the result with the proxy could also be passed if the proxy uses the same sequence number for both connections but it is a remote case (about 1 over 2^{32});
- **TLSKeysTest**: either if the proxy is used or not, the final verdict is pass because the proxy, thanks to the CONNECT method, allows the creation of a end-to-end TLS session;
- **SmallTTLTest**: the verdict is always pass even in presence of a proxy since in the portion of network connecting the client and the server or the client and the proxy there is no device which resets the TTL value; the testcase has not been performed by swapping client and server since the either the server and the proxy are located on the same LAN and then they are directly connected;
- **MD5Test**: the results are reasonably predictable: pass without proxy and fail with the proxy because the HTTP headers such as required host, Host and Via are modified;
- **XForwardedForTest**: in both cases the testcase verdict is pass because the proxy does not introduce any variation to the X-Forwarded-For header.

Chapter 6

Conclusion

E-WASSI aims to introduce in the OT and IT worlds such devices called Guardians in order to continuously assess and protect all the network devices that are due to protect the network itself. The proposed implementation is only a small portion of what the whole project represents and it establishes a starting point for the future works. TTCN-3 combined with Titan turned out to be very practical for the architecture designed for E-WASSI and for developing test cases, actual representation of the protocol WASSI talked by the Guardians, which are the core of project. TTCN-3 is a very powerful weapon thanks to all the constructs and mechanisms provided to develop test cases. Titan, which is the tool that makes the TTCN-3 abstract test suite an executable program, provides a very large developing and execution environment thanks to its implementations and extensions that improve TTCN-3 Core language and balance its own limitations. The few test cases that have been developed showed that in a secure environment such the network implemented in GNS3 and used for the evaluation step the verdict is in most cases the expected one.

E-WASSI in the version proposed in this work is not complete since not all its missions have been developed and implemented and even the "Uncover" and "Assess" missions can be improved by adding new test cases. For instance a new test case could leverage the logic behind the *traceroute* program after that the FragmentationTest failed in order to discover who reassembled the fragments: one Guardian sends multiple pairs of packets with incremental Time To Live as happens in *traceroute*. In such a way the sender Guardian can monitor whenever the ICMP Time exceeded packets are not received in pairs. This procedure can reveal the IP address or at least the subnet associated to the reassembler. Another important step that has not been treated in this work is the opportune choice of the test cases or campaigns: when and why a Guardian should prefer a campaign rather than another one? A final step that must be done once that E-WASSI is completed could be the creation of a community of users and developers that can implement their own modules in order to enrich the already existing functionalities of the Guardians.

In conclusion, the work done so far shows all the potential of E-WASSI, despite the implementation is far away from the final shape that the project must adopt. The ultimate solution proposed by E-WASSI can become an always-improving milestone addressing a very delicate field which so far does not own a proper ad-hoc solution.

Bibliography

[1] F. Kargl, R. W. van der Heijden, H. König, A. Valdes, and M. C. Dacier, "Insights on the Security and Dependability of Industrial Control Systems", *IEEE Security Privacy*, vol. 12, no. 6, pp. 75–78, 2014. DOI: 10.1109/MSP.2014.120.

[2] M. Dacier, F. Kargl, H. König, and A. Valdes, "Network attack detection and defense: securing industrial control systems for critical infrastructures", *Informatik Spektrum*, vol. 37, no. 6, pp. 605–607, 2014.

[3] E. Bertino and N. Islam, "Botnets and Internet of Things Security", *Computer*, vol. 50, pp. 76–79, Feb. 2017. DOI: 10.1109/MC.2017.62.

[4] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "DDoS in the IoT: Mirai and Other Botnets", *Computer*, vol. 50, no. 7, pp. 80–84, 2017. DOI: 10.1109/MC.2017.201.

[5] M. Paquet-Clouston, O. Bilodeau, and D. Décary-Hétu, "Can We Trust Social Media Data?: Social Network Manipulation by an IoT Botnet", Jul. 2017, pp. 1–9. DOI: 10.1145/3097286.3097301.

[6] S. Soltan, P. Mittal, and H. V. Poor, "BlackIoT: IoT botnet of high wattage devices can disrupt the power grid", in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 15–32.

[7] N. Falliere, L. O. Murchu, and E. Chien, "W32. stuxnet dossier", *White paper, Symantec Corp., Security Response*, vol. 5, no. 6, p. 29, 2011.

[8] L. Xing, "Cascading Failures in Internet of Things: Review and Perspectives on Reliability and Resilience", *IEEE Internet of Things Journal*, vol. 8, no. 1, pp. 44–64, 2021. DOI: 10.1109/JIOT.2020.3018687.

[9] E. Guttman, "Autoconfiguration for IP networking: enabling local communication", *IEEE Internet Computing*, vol. 5, no. 3, pp. 81–86, 2001. DOI: 10.1109/4236.935181.

[10] S. Cheshire and M. Krochmal, "Multicast DNS", RFC 6762, February, Tech. Rep., 2013.

[11] B. Aboba, D. Thaler, and L. Esibov, "Link-local multicast name resolution (LLMNR)", RFC 4795 (Informational), Internet Engineering Task Force, Tech. Rep., 2007.

[12] J. Garofalakis, Y. Panagis, E. Sakkopoulos, and A. Tsakalidis, “Web service discovery mechanisms: Looking for a needle in a haystack”, in *International Workshop on Web Engineering*, vol. 38, 2004, p. 25.

[13] S. Cheshire and M. Krochmal, “DNS-based service discovery”, RFC 6763, February, Tech. Rep., 2013.

[14] S. Albright, P. J. Leach, Y. Gu, Y. Y. Goland, and T. Cai, “Simple Service Discovery Protocol/1.0”, Internet Engineering Task Force, Internet-Draft draft-cai-ssdp-v1-03, Nov. 1999, Work in Progress, 18 pp. [Online]. Available: `https://datatracker.ietf.org/doc/html/draft-cai-ssdp-v1-03`.

[15] C.-F. Lai, Y.-M. Huang, and H.-C. Chao, “DLNA-based multimedia sharing system for OSGI framework with extension to P2P network”, *IEEE Systems Journal*, vol. 4, no. 2, pp. 262–270, 2010.

[16] M. Villari, A. Celesti, M. Fazio, and A. Puliafito, “AllJoyn Lambda: An architecture for the management of smart environments in IoT”, in *2014 International Conference on Smart Computing Workshops*, 2014, pp. 9–14. DOI: `10.1109/SMARTCOMP-W.2014.7046676`.

[17] C. Wagner, A. Dulaunoy, G. Wagener, and A. Iklody, “MISP: The Design and Implementation of a Collaborative Threat Intelligence Sharing Platform”, in *Proceedings of the 2016 ACM on Workshop on Information Sharing and Collaborative Security*, ACM, 2016, pp. 49–56.

[18] V. Hassija, V. Chamola, V. Saxena, D. Jain, P. Goyal, and B. Sikdar, “A survey on IoT security: application areas, security threats, and solution architectures”, *IEEE Access*, vol. 7, pp. 82 721–82 743, 2019.

[19] T. Aste, P. Tasca, and T. Di Matteo, “Blockchain Technologies: The Foreseeable Impact on Society and Industry”, *Computer*, vol. 50, no. 9, pp. 18–28, 2017. DOI: `10.1109/MC.2017.3571064`.

[20] O. Novo, “Blockchain Meets IoT: An Architecture for Scalable Access Management in IoT”, *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 1184–1195, 2018. DOI: `10.1109/JIOT.2018.2812239`.

[21] P. Lv, L. Wang, H. Zhu, W. Deng, and L. Gu, “An IOT-Oriented Privacy-Preserving Publish/Subscribe Model Over Blockchains”, *IEEE Access*, vol. 7, pp. 41 309–41 314, 2019. DOI: `10.1109/ACCESS.2019.2907599`.

[22] U. Javaid, M. N. Aman, and B. Sikdar, “Blockpro: Blockchain based data provenance and integrity for secure iot environments”, in *Proceedings of the 1st Workshop on Blockchain-enabled Networked Sensor Systems*, 2018, pp. 13–18.

[23] Y. Yu, Y. Li, J. Tian, and J. Liu, “Blockchain-based solutions to security and privacy issues in the Internet of Things”, *IEEE Wireless Communications*, vol. 25, no. 6, pp. 12–18, 2018.

[24] U. Javaid, M. N. Aman, and B. Sikdar, “DrivMan: Driving trust management and data sharing in VANETS with blockchain and smart contracts”, in *2019 IEEE 89th Vehicular Technology Conference (VTC2019-Spring)*, IEEE, 2019, pp. 1–5.

[25] T. M. Fernández-Caramés and P. Fraga-Lamas, “A Review on the Use of Blockchain for the Internet of Things”, *IEEE Access*, vol. 6, pp. 32 979–33 001, 2018. DOI: 10.1109/ACCESS.2018.2842685.

[26] K. Valtanen, J. Backman, and S. Yrjölä, “Blockchain-Powered Value Creation in the 5G and Smart Grid Use Cases”, *IEEE Access*, vol. 7, pp. 25 690–25 707, 2019. DOI: 10.1109/ACCESS.2019.2900514.

[27] U. Javaid, A. K. Siang, M. N. Aman, and B. Sikdar, “Mitigating loT device based DDoS attacks using blockchain”, in *Proceedings of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems*, 2018, pp. 71–76.

[28] K. R. Ozyilmaz and A. Yurdakul, “Designing a Blockchain-based IoT with Ethereum, swarm, and LoRa: the software solution to create high availability with minimal security risks”, *IEEE Consumer Electronics Magazine*, vol. 8, no. 2, pp. 28–34, 2019.

[29] V. Sharma, “An energy-efficient transaction model for the blockchain-enabled internet of vehicles (IoV)”, *IEEE Communications Letters*, vol. 23, no. 2, pp. 246–249, 2018.

[30] H. M. Hamad and M. Al-Hoby, “Managing intrusion detection as a service in cloud networks”, *International Journal of Computer Applications*, vol. 41, no. 1, 2012.

[31] S. Chandrasekhar and M. Singhal, “Efficient and scalable query authentication for cloud-based storage systems with multiple data sources”, *IEEE Transactions on Services computing*, vol. 10, no. 4, pp. 520–533, 2015.

[32] Q. Jiang, J. Ma, F. Wei, Y. Tian, J. Shen, and Y. Yang, “An untraceable temporal-credential-based two-factor authentication scheme using ECC for wireless sensor networks”, *Journal of Network and Computer Applications*, vol. 76, pp. 37–48, 2016.

[33] P. Hu, H. Ning, T. Qiu, H. Song, Y. Wang, and X. Yao, “Security and privacy preservation scheme of face identification and resolution framework using fog computing in internet of things”, *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1143–1155, 2017.

[34] C. Li, Z. Qin, E. Novak, and Q. Li, “Securing SDN infrastructure of IoT–fog networks from MitM attacks”, *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1156–1164, 2017.

[35] C.-K. Chu, S. S. Chow, W.-G. Tzeng, J. Zhou, and R. H. Deng, “Key-aggregate cryptosystem for scalable data sharing in cloud storage”, *IEEE transactions on parallel and distributed systems*, vol. 25, no. 2, pp. 468–477, 2013.

[36] P. Rizomiliotis and S. Gritzalis, “ORAM based forward privacy preserving dynamic searchable symmetric encryption schemes”, in *Proceedings of the 2015 ACM Workshop on Cloud Computing Security Workshop*, 2015, pp. 65–76.

[37] M. Naveed, M. Prabhakaran, and C. A. Gunter, “Dynamic searchable encryption via blind storage”, in *2014 IEEE Symposium on Security and Privacy*, IEEE, 2014, pp. 639–654.

[38] B. Cavallo, G. Di Crescenzo, D. Kahrobaei, and V. Shpilrain, "Efficient and secure delegation of group exponentiation to a single server", in *International workshop on radio frequency identification: security and privacy issues*, Springer, 2015, pp. 156–173.

[39] K Pavani and A Damodaram, "Intrusion detection using MLP for MANETs", 2013.

[40] R. V. Kulkarni and G. K. Venayagamoorthy, "Neural network based secure media access control protocol for wireless sensor networks", in *2009 international joint conference on neural networks*, IEEE, 2009, pp. 1680–1687.

[41] L. Xiao, C. Xie, T. Chen, H. Dai, and H. V. Poor, "A mobile offloading game against smart attacks", *IEEE Access*, vol. 4, pp. 2281–2291, 2016.

[42] L. Xiao, Q. Yan, W. Lou, G. Chen, and Y. T. Hou, "Proximity-based security techniques for mobile users in wireless networks", *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 12, pp. 2089–2100, 2013.

[43] L. Xiao, Y. Li, G. Han, G. Liu, and W. Zhuang, "PHY-layer spoofing detection with reinforcement learning in wireless networks", *IEEE Transactions on Vehicular Technology*, vol. 65, no. 12, pp. 10 037–10 047, 2016.

[44] M. Ozay, I. Esnaola, F. T. Y. Vural, S. R. Kulkarni, and H. V. Poor, "Machine learning methods for attack detection in the smart grid", *IEEE transactions on neural networks and learning systems*, vol. 27, no. 8, pp. 1773–1786, 2015.

[45] C. Shi, J. Liu, H. Liu, and Y. Chen, "Smart user authentication through actuation of daily activities leveraging WiFi-enabled IoT", in *Proceedings of the 18th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, 2017, pp. 1–10.

[46] L. Xiao, X. Wan, X. Lu, Y. Zhang, and D. Wu, "IoT security techniques based on machine learning: How do IoT devices use AI to enhance security?", *IEEE Signal Processing Magazine*, vol. 35, no. 5, pp. 41–49, 2018.

[47] L. Xiao, X. Wan, and Z. Han, "PHY-layer authentication with multiple landmarks with reduced overhead", *IEEE Transactions on Wireless Communications*, vol. 17, no. 3, pp. 1676–1687, 2017.

[48] Z. Yan, P. Zhang, and A. V. Vasilakos, "A survey on trust management for Internet of Things", *Journal of network and computer applications*, vol. 42, pp. 120–134, 2014.

[49] C. Li and G. Wang, "A light-weight commodity integrity detection algorithm based on Chinese remainder theorem", in *2012 IEEE 11th international conference on trust, security and privacy in computing and communications*, IEEE, 2012, pp. 1018–1023.

[50] Y. Rouf, M. Shtern, M. Fokaefs, and M. Litoiu, "A hierarchical architecture for distributed security control of large scale systems", in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, IEEE, 2017, pp. 118–120.

[51] M. Krichen and R. Alroobaea, “A new model-based framework for testing security of IoT systems in smart cities using attack trees and price timed automata”, in *14th International Conference on Evaluation of Novel Approaches to Software Engineering*, SCITEPRESS-Science and Technology Publications, 2019, pp. 570–577.

[52] B. Kordy, L. Piètre-Cambacédès, and P. Schweitzer, “DAG-based attack and defense modeling: Don’t miss the forest for the attack trees”, *Computer science review*, vol. 13, pp. 1–38, 2014.

[53] R. Kumar, E. Ruijters, and M. Stoelinga, “Quantitative attack tree analysis via priced timed automata”, in *International Conference on Formal Modeling and Analysis of Timed Systems*, Springer, 2015, pp. 156–171.

[54] L. Fawcett, S. Scott-Hayward, M. Broadbent, A. Wright, and N. Race, “Tennison: A distributed SDN framework for scalable network security”, *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 12, pp. 2805–2818, 2018.

[55] *Zeek official website*, https://zeek.org/, Accessed: 27-01-2021.

[56] Cisco, *Snort official website*, https://www.snort.org/, Accessed: 27-01-2021.

[57] Open Networking Foundation, *ONOS official website*, https://opennetworking.org/onos/, Accessed: 27-01-2021.

[58] InMon, *sFlow-RT official website*, https://sflow-rt.com/, Accessed: 27-01-2021.

[59] L. Krämer, J. Krupp, D. Makita, T. Nishizoe, T. Koide, K. Yoshioka, and C. Rossow, “Amppot: Monitoring and defending against amplification ddos attacks”, in *International Symposium on Recent Advances in Intrusion Detection*, Springer, 2015, pp. 615–636.

[60] DutchSec, *Honeytrap git repository*, https://github.com/honeytrap/honeytrap, Accessed: 27-01-2021.

[61] *nmap official website*, https://nmap.org/, Accessed: 27-01-2021.

[62] L. Deri and S. Suin, “Effective traffic measurement using ntop”, *IEEE Communications Magazine*, vol. 38, no. 5, pp. 138–143, 2000.

[63] E. Lear, R. Droms, and D. Romascanu, “Manufacturer Usage Description Specification”, RFC 8520, March, Tech. Rep., 2019.

[64] E. Cooke, M. Bailey, D. Watson, F. Jahanian, J. Nazario, and A. Networks, “The Internet motion sensor: A distributed global scoped Internet threat monitoring system”, Feb. 2021.

[65] T. Ostrand, “Black-Box Testing”, in. Jan. 2002, ISBN: 9780471028956. DOI: 10.1002/0471028959.sof022.

[66] T. Csöndes, J.Z. Szabó, R. Gecse, P. Krémer, C. Koppány, G. Réthy, F. Bozóki, G. Adamis, *TTCN-3 Course Presentation Material*, http://www.ttcn-3.org/files/Titan_TTCN3_course.pdf, Accessed: 27-01-2021.

[67] Wikipedia, *Test suite*, https://en.wikipedia.org/wiki/Test_suite, Accessed: 27-01-2021.

[68] ETSI, *TTCN-3 Introduction*, http://www.ttcn-3.org/index.php/about/introduction, Accessed: 27-01-2021.

[69] "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3", ETSI, ES 201 873, 2000.

[70] "Testing and Test Control Notation version 3", ITU-T, ITU-T Z.160 series, 2007.

[71] "Conformance Test Specification for SIP (IETF RFC 3261)", ETSI, TS 102 027, 2006.

[72] "Conformance Testing for the Network layer of HiperMAN/WiMAX terminal devices", ETSI, TS 102 624, 2009.

[73] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock, "An introduction to the testing and test control notation (TTCN-3)", *Computer Networks*, vol. 42, pp. 375–403, Jun. 2003. DOI: 10.1016/S1389-1286(03)00249-4.

[74] ETSI, *Titan toolset*, http://www.ttcn-3.org/index.php/tools/16-tools-noncom/112-non-comm-titan, Accessed: 27-01-2021.

[75] J. Szabó and T. Csöndes, "TITAN, TTCN-3 test execution environment", 2007.

[76] Eclipse, *titan.core*, https://github.com/eclipse/titan.core, Accessed: 27-01-2021.

[77] ——, *The Run-time Configuration File*, https://github.com/eclipse/titan.core/blob/5e2eb8ee1038a19f1f2817d2eb87be28400e9b3f/usrguide/referenceguide/7-the_run-time_configuration_file.adoc, Accessed: 27-01-2021.

[78] ——, *TTCN-3 Language Extensions*, https://github.com/eclipse/titan.core/blob/master/usrguide/referenceguide/4-ttcn3_language_extensions.adoc, Accessed: 27-01-2021.

[79] E. Rescorla, "Keying material exporters for transport layer security (tls)", RFC 5705, March, Tech. Rep., 2010.

[80] Free Software Foundation, *tinyproxy*, https://github.com/tinyproxy/tinyproxy, Accessed: 27-01-2021.