Automatic Test Framework Anomaly Detection in Home Routers

Pavel Pogosov

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Supervisors

Prof. Tuomas Aura

Prof. Colin Boyd

Aalto University
School of Science

NTNU
Department of Information Security and Communication Technology

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Pavel Pogosov

Title Automatic Test Framework Anomaly Detection in Home Routers

Degree programme Security and Mobile Computing (NordSecMob)

Major Security and Mobile Computing **Code of major** T3011

Supervisor and advisor Prof. Tuomas Aura, Prof. Colin Boyd

Date 20/7/2020 **Number of pages** 56+1 **Language** English

Abstract

In a modern world most people have a home network and multiple devices behind it. This devices include simple IoT, that require external protection not to join a botnet. This protection can be granted by a security router with a feature of determining the usual network traffic of a device and alerting its unusual behaviour. This work is dedicated to creating a testbed to verify such router's work. The test bed includes tools to capture IoT traffic, edit and replay it. Created tool supports UDP, TCP, partially ICMP and is extendable to other protocols. UDP and TCP protocols are replayed using OS sockets at transport network layer. The methods described have proved to work on a real setup.

Keywords security router, traffic replay, testbed, testbench, network protection, IoT protection

Preface

The master thesis project was conducted and financially supported by F-Secure corporation in Helsinki in 2018. The software developed during the work is the property of F-Secure. The purpose of the work is to end-to-end test the company's network protecting devices reaction on traffic.

I would like to express gratitude to Alexey Kirichenko for picking me at HIAC meeting, Jari Lehtonen managing my work, Jussi Kallio supervising it.

A great support was given by my supervisor professor Tuomas Aura and Aalto University giving me place and more means for running experiments. His impact on this paper is comparable to my own.

I would also like to thank my friend Ricardo Vietez Parra for a great deal of advice and help.

Contents

1

Introduction

In the 21st century, most people have a home wireless network. The network is built around a gateway router connected to a provider network and can include access points, connected personal computers, mobile phones, and often autonomous appliances connected to Internet — Internet of things (IoT) devices. The IoT devices are usually relatively simple gadgets fulfilling a narrow function and exchanging data with a server in the cloud. These devices, in order to be produced cheaper and simpler, often lack security. This makes them a perfect target for a hacker attack.

Network devices can be protected by a firewall, but it does not give an absolute guarantee against attacks, and the IoT devices can be infected in some way. In order to detect infected devices, some security-oriented routers have a feature for traffic anomaly detection. These routers observe the connected devices' behaviour, and their algorithms determine the types of the devices by the traffic they produce. After that, they detect behavioural anomalies, raise alarms, and may react to prevent the possible intrusion.

Anomaly detection, like any Intrusion Detection System (IDS), can produce errors: false positive or false negative results. In order to evaluate such feature implementation and to reduce the error rate, this function needs testing facility.

This thesis is dedicated to the creation of a testbed which simulates connected device behaviour in order to test the IDS functionality in a security router. The security router tested in the created testbed will be called *router under test* (RUT). The typical use case of the testbed is to run an experiment with an IoT device connected to a security router, let the router identify the gadget, and afterwards modify a part of the traffic like a destination IP address and see how the router reacts. Ideally, the IoT device is used in the testbed only once — to record a traffic trace. Then, all the later operations are done by replaying the saved traffic traces with possible modifications.

More specifically, the thesis has the following goals:

- Understand the requirements and methods for replaying network traffic in a test lab.
- Design a testbed for security routers. Develop the process and tools working within that testbed that allows easy recording of the behaviour of an arbitrary IoT device, modify the trace, and replay it to a RUT.
- Verify the created testbed produces the expected results on a real RUT and a real IoT device. This means that the RUT reacts to the replayed traffic in same manner as to the real IoT device.
- Modify the recorded traffic trace in various ways and verify that the expected anomaly detection alerts are triggered.

In order to achieve these goals, it was decided to design a testbench working with physical RUT and IoT devices. The testbench consists of a computer operating the test process and auxiliary network devices. The testbed application is implemented as a set of command-line applications. Various open source utilities and libraries were engaged to manipulate the recorded traces. The IoT device was simulated with a help of virtual device instances utilizing stored traces.

The behaviour of an IoT device is identified by the network traffic it produces and is primarily stored as network traces.

The testbed developed in this work was proved to be working by making a RUT identify simulation as a real device. It also triggered an anomaly alert after modifying the traffic.

The rest of the thesis paper is organised as follows. Background chapter describes reasons for a project to be initiated, gives an overview on basic network technologies, surveys existing protection solutions and existing testbeds for them. Network traffic analysis chapter tells about theoretical approach to analyse the substance the testbed is working with. Testbench overview chapter describes basic principles of the project that has been implemented and the testbench itself from a perspective of a person operating it. Upper level implementation describes software architecture and tools used for the project. The next Protocol replay implementation chapter describes how particular network protocols replaying components work. The last Discussion chapter covers topics that raise doubts, project results, potential future work on the project and alternative solutions that could take place.

2

Background

This chapter considers some aspects and methods of network protection. It also overviews some of the existing tools with a similar purpose to the developed one in this work.

2.1 Traffic traces

A basic unit of traffic is a packet. World wide used TCP/IP protocols form a packet as a header section containing a protocol related service data, followed payload section with any information carried within the protocol. For example, a UDP header contains source and destination IP addresses and port numbers. A payload can contain another protocol data, thus one protocol wrapping another, creating a protocol stack. A TCP/IP model considers a stack of physical, link, transport and application layer.

Most traffic can be divided into sessions — a series of packets related to each other, handling one process of a certain application. It can be tracked by particular protocol data-field, for example, transport layer protocol IP addresses and ports. IP address is used to refer to a host interface in a network. Typical transport protocols TCP and UDP have source and destination ports. Those are numbers that help OS to understand to which application received packet should be transferred. Clients usually generate a random port number for each session for themselves, where as servers port is fixed in order for client to know it, however, it is not mandatory. This way a session is defined by source IP address, source port, destination IP address and destination port.

2.1.1 Recording

Captured internet traffic is mostly stored in pcap files. The most famous utility for capturing traffic on Linux is tcpdump [1]. It is a command line tool that can just show a summary of a traffic or save the whole trace to a pcap file; allows to filter traffic, for instance by host IP; gives opportunity to select network interface to listen. There are also widely used applications with a friendly UI like Wireshark or Fiddler, allowing user to capture traffic, look through parsed packets and save it. From low level perspective, there are numerous libraries allowing traffic sniffing. In this project *scapy* library was used for sniffing, along with packets modifying and sending.

2.1.2 Replaying

Whereas sending one packet into a network is simple — you can just use some library to utilize OS socket, replaying traffic through a network is complicated. Protocol data from the captured traces may differ from the state required by new network conditions. One of the first encountered problems was TCP/UDP destination port. The RUT tested in lab along with most home used routers uses network address translation (NAT) technology. NAT separates router's LAN and WAN so that when a packet goes from LAN to WAN the original source IP address is changed to another IP address. The original purpose of NAT was to avoid IPv4 network address exhaustion, since IPv4 protocol supports a limited amount of addresses equal to 2^{32} in total excluding loopback, broadcast and other reserved addresses. The most widely used type of NAT is one-to-many NAT that is mapping all LAN addresses into a router WAN IP address. This NAT type supporting router stores the ongoing connections mapping from source address and source port number into a new source port. Source port number might remain the same, but can also be changed, for example to avoid collisions of same source ports from different LAN hosts. Port changing depends on a particular router software implementation making the port mapping impossible to predict for a general case. These port numbers should be consistent throughout a session replay. Other typical packet parameters that need careful consideration, when replaying traffic, will be discussed in the next chapter.

2.2 Reasons and motivation for home network protection

An IoT device is often done simple (relieved from directly unnecessary features) and lacks security making it an easy prey for a hacker. And adversaries do actively seek a way to take control of it [2].

Security routers are being developed in order to protect home network devices. Some of them have a feature of traffic anomaly detection, that detects device abnormal behaviour. It might be questioned at this point, why we assume a device to be infected instead of improving protections that would prevent the infection in the first place. There are many ways to infect a device. Protecting it from outbound connection is not enough because an attack can come from inside the home network, from malware installed in a PC for example. Isolating an IoT device from any other devices will not work if an attacker acts physically and installs malware to a device manually. After all, a device may have malware ever since it has been produced! That is why it is important to make the assumption that no protection can work absolutely as the first line of defence and the device can be infected.

A common security question arises, what motivates the adversary to attack. Most hackers are not concerned with a regular person's bedroom temperature. The reason why he would like to control someone's thermostat is probably to make it join his botnet. A botnet is a group of infected end-hosts under the command of a bot-master [3]. According to [2], an overwhelming majority of bots are IoT devices. Bots can be used for computational power (for example for cryptocurrency mining), or as source of attack to other systems.

There are several possible reasons for an ordinary person to try to protect his IoT devices. One of them is the home network being flooded due to an intensive DDoS attack [2], slowing down the internet connection, or even disrupting device from functioning normally. Intensive use of device computational power can lead to a significant increase of electricity consumption. Another reason to protect oneself is possibility of a gadget spoiling one's life in its way. This can happen while being either under personal attack by ill-wisher, or a person is just a victim of a mass attack causing thermostats to raise the indicator data so that a lot of people catch cold, in order to flood hospitals or paralyze society. A DDoS attack launched from home, might cause the IP address to be blacklisted in a certain segment of Internet, as black lists of IP addresses exist and can be used by server providers[4].

2.3 IoT devices attacks

One possible way to infect an IoT device is to attempt to connect to it using default vendor default credentials via SSH[5], which are often neglected to be changed by users. Vulnerability is widely introduced utilizing telnet protocols for remote administration of devices[5]. Yet again attackers scan network for those devices and attempt to brute force credentials. Backdoors left by vendors or malware is another point to tempt an attacker.

To try to connect to an IoT, malware should know the host address and an SSH port. To find that out port scanning is performed. For example, for TCP protocol it can be an attempt to start TCP three-way handshake. If the host exists at the tried address and there is an application listening to the port, it will answer with SYN-ACK packet on the scanner SYN message. If scanned address does not exist or port is not listened to, an ICMP message can be sent informing that destination is unreachable.

Another way to hijack a device is using man in the middle (MITM) attack. MITM concept means intercepting communication between two instances and posing malicious intermediate system as original sender to both parties. This attacks can work against devices that do not have digital certificates that would authorize a server they are trying to access. For example, when IoT device tries to update software, MITM can spoof the updates to ones containing malware thus taking over control of a device.

2.4 Firewall

The first line of defence in a network is a firewall. Basic function of a firewall is filtering packets to go through it based on defined rules. This basic feature can prevent attackers from scanning or connecting to devices in your network.

2.4.1 Packet filtering

Firewall rules can be applied to a certain connection direction (inbound or outbound connections) and are determined with connection 5-tuple: source host IP address (also possible to put a subnet range), source port, destination IP address, destination port and protocol used. Also other transport layer header parameters can be potentially used.

It is usually a good idea (unless you have hosts serving external clients) to forbid all inbound (incoming from outside) connections for your home network to prevent adversaries from scanning your network or connecting to your devices. Filtering outbound connections should be done carefully to allow connected devices function normally. Usually, certain applications use certain transport layer ports. There are hundreds of applications utilising different ports, the most commonly used protocols and ports are the described further: HTTP and HTTPS use TCP ports 80 and 443; SSH used for secure connection to a host usually uses TCP port 22; Ports 20 and 21 are used for FTP; SMTP utilises port 25. It should be kept in mind that these are just conventional port usages, nothing prevents a server application from listening to a different port. It makes sense to filter outbound traffic based on those services usage, for example it would be very suspicious if an IoT device connects somewhere by SSH. Ports 80 or 443, however are typically used by all devices.

Firewalls simple setup and properly defined rules would protect a LAN from attacks by forbidding connections to your devices from outside and blocking certain protocols. An example of a protocol that can be blocked is IRC (typical way to send commands from botmaster) using ports 6666-6669 [3]. Firewalls can also stop malware scanning the network for vulnerable IoT devices [6].

2.4.2 Stateful packet filtering

Stateful packet filtering is a technology improving the basic packet filtering. It ensures that the packet flow corresponds to a specified protocol. The first packet of session is checked against firewall rules and a session is cached. All subsequent packets of the session are checked against the saved cache [7]. After final packet is received by the firewall, no more packets for this session are allowed to go through. This boosts performance and mitigates the simple NAT breach. The problem of stateless NAT is that is does not know when a session has ended and keeps the port mapping alive for a certain timeout. This allows incoming packets with corresponding ports to go through for some while.

2.4.3 Deep packet inspection

Not only the transport layer is considered by firewall. A new generation of firewalls is being developed capable of dealing advanced threats [8]. With deep packet inspections (DPI) the firewalls can be capable of Intrusion Prevention System (IPS), Virtual Private Network (VPN), Anti-Spam, Anti-Virus and URL Filtering while optimising performance and throughput.

Pattern or signature matching is the primary approach used firewalls with IDS features. This function analyzes each packet against a database of known network attacks [9]. This is a very difficult problem, because network activity requires a high performance, where as function implementation needs to check data against a very large signature database. DPI algorithms include regular expression matching, string matching algorithms, Wu-Manber algorithm, SBOM and others. DPI can be classified as signature based, application layer based and behaviour based identification.

2.4.4 Firewall problems

In past decade significant increase of https usage for transmitting web data is observed. Encrypted connections make DPI impossible. From firewall perspective it can have two solutions: forbidding https connections or serving as a proxy server to decrypt and reencrypt connection [10] using own certificates. From user perspective, using insecure connection is unacceptable in most cases like online banking or even social network usage, since it exposes everything: passwords, tokens, personal information.

Stateful packet inspection cannot work properly with mobile devices changing their networks. Since connection interruption can happen at any moment without a warning, the NAT port remains open allowing an attacker to continue the session. Firewall should also have an improved implementation for protocols like Multipath TCP **multipath-tcp-old**

DPI approach is comparatively very expensive — it requires a large computational power, is much slower a simple firewall significantly reducing network gateway throughput.

Overall, it may happen that an adversary slips past the firewall via allowed connections, through an encrypted tunnel or simply infects a device in some other way. The internal network usually does not have restrictions, and thus one exposed device can infect the others via network. That is where intrusion detection systems (IDS) can be useful.

2.5 Intrusion detection system

Another protection method that can be used is monitoring the network, remembering the devices' behaviour and taking precautions (like informing a user or banning the device) when a certain device behaves in a way it is not expected to. In a home network this work can be done by a security router with an Intrusion Detection System (IDS) feature that is going to be tested in this work.

Classifying traffic as malicious can rely on deep packet inspection or monitoring the traffic properties. A signature or rule based approach works for known attacks. A machine learning approach allows an IDS to be more flexible and detect anomalies not caught by the previous approaches.

There are two approaches for intrusion detection: network based and host based IDS. Network based IDS dynamically scans network traffic checking for anomalies or comparing against malicious signatures database. Host based IDS are installed on the machines directly. The latter tools monitor system state for left backdoors, certain segments of memory changes, network-based logs. This work focuses on network based IDS.

2.6 Intermediate access device

Another approach to protect vulnerable IoT devices is to connect them to a well protected computer[11] rather than directly to the home network. This will ensure that the devices can be directly accessed only after a strong authentication process. Such an approach will limit attack surface only to existing outgoing connections from those devices by man in the middle method. However, this method makes sense if those device are actually meant to be accessed from anywhere in internet, so it should be applied to certain subset of IoT devices.

There are IoT devices that are not suitable for this approach. Those maintain a permanent live connection to its manufacturer servers and are meant to be accessed from there. However that kind of devices should have certificates set in order to be safe from man in the middle attacks.

Other intermediate devices are called smart hubs. Those are produced industrially but position themselves based on comfort rather than security, although security is one of the primary aspects of those products. The basic concept is to connect all IoT devices to one device that controls them, so that user uses just one "simple" application to set up his smart house. There are several possible ways to connect IoT devices to a smart hub including Bluetooth LE, Wi-Fi, Z-Wave and ZigBee. Smart Home and City are examples of such products [12]. IoT devices and smart hubs utilize strong identity mechanism to authorize each other.

2.7 Testing network equipment

There are many aspects of network equipment that can be tested. One aspect is physical properties of a device. Users are primarily interested by a device bandwidth — maximum speed of data transfer, other properties include power efficiency or signal quality [13]. Another aspect for a security router is functional testing — how well it fulfills a security check without a stress load.

There are many ways to organise development: agile, continuous, spiral, waterfall and other methodologies. But in any development process testing is a crucial step and required for a successful release. Test automation in a pipeline is a clue to ensure successful continuous delivery of a product.

Testing can happen on different levels: unit tests check a simple component's work; integration tests check connected components mutual work; acceptance or end-to-end testing checks if a whole system is capable of handling a use case. The work described in this paper was about end-to-end testing a security router.

From a system awareness perspective, testing can be categorized as white, black and grey box testing. For a white box testing [14], tester should know how the tested code works in order to create tests covering all execution statements or paths. It is usually applied in a lower test levels (unit, integration). Black box approach means that testers don't know how the tested unit is working. In this case test plan is created based on functional requirements and needs to cover test cases. Grey box testing is a hybrid one between previous two. In my work a black box testing was performed, as my knowledge about the router was mostly limited to communication and data extraction ways.

For the work done on the described project, there are two important test case sets. First one is to modify traffic in such a way that it represents actually abnormal behaviour, so that we are expecting the RUT to trigger an alert. Another case is to test possible false positive anomaly detection. This can happen if the IoT cloud service changes its IP address, for example.

2.8 Similar existing solutions overview

Numerous tools for testing network devices exist focusing on different purposes. To my knowledge, none of them were aiming to simulate a particular device behaviour. However, those tools have a number of good practices to learn.

2.8.1 Tcpreplay

A good tool to send traffic from pcap files through a device interfaces is *tcpreplay*[15]. This program is working at network layer 2, sending the packages from a file one by one with a user defined speed.

In addition to *tcpreplay*, there is a tool *tcprewrite* that allows to change MAC addresses, IP addresses and ports in pcap files and also other parameters.

This tool is very deterministic, but the problem is that it is static. In a general case one cannot predict NAT port translation by a router under test thus RUT can have a different NAT port mapping than in the trace. Lost packets for TCP is another case where this tool would work incorrectly because the RUT may drop different packets than the original router did, and *tcpreplay* does not consider the current simply replaying packets from orignial trace as is.

2.8.2 Stateful traffic replay for web application proxies

This article[10] reviews several existing solutions and approaches and offers a new one for testing a non-transparent security proxy. The key goal of the tool described is to precisely reconstruct the application layer payload.

The possible target of benchmarking by the tool is a web application firewall (WAF). WAF does deep packet inspection searching for possible attacks, for example, cross-site-scripting for web pages, or viruses hidden in downloaded files. However, deep packet inspection is not possible on encrypted traffic. That is why advanced security gateways dealing with this problem work as non-transparent proxy servers decrypting the flow and encrypting it when transferring data further. Here is where the described tool comes into business in this article.

The work focuses on protocol modifications to a suit proxy, functional behaviour to suit different types of non-transparent proxy devices, maintaining concurrent connections, and making connections resistant to network like packets lost, packets out-of-order etc. Memory efficiency was one of the targets of this project, memory pool used by the tool has a limited size, and replayed data is loaded not exceeding it. The work also handles DNS requests to be correct by emulating the DNS server for the device under test.

In order to save resources, the system reads the proxy's response to understand whether the connection needs to be maintained on each side (LAN and WAN) of the network. For concurrent mode, the presented tool keeps the original order of replayed sessions, but the packet timing is left undetermined.

2.8.3 Marine Corps Tactical Systems Support Agency stateful TCP replay

The work[16] is dedicated to creating a tool for stress testing WAN by loading it with a stateful TCP replay. This tool is aimed to make the tests repeatable. The work focuses on TCP and testing WAN traffic optimisation.

The tool in this work acts in a server-client manner. It is able to modify a trace to fit the test environment and maintain inter-packet timing intervals.

The program implemented for that theses consists of two parts: preparation and replaying. The preparing module parses a pcap file into a database, removes repeating packets and defines the timing. The second module is responsible for listening to incoming packets and sending those belonging to the corresponding host along with keeping the timing.

The program is designed to keep the time intervals between packets. However, the timing would not be accurate in case of concurrent flows in both directions, due to a constraint that an expected packet in order will not be sent until the previous one has arrived.

The machines used for running tests are set up to drop incoming packets for the corresponding TCP sessions in order to force the OS to ignore them not to interfere into the replaying process.

This work also provides methods for comparison between original and replayed trace.

2.8.4 TCPopera

Another tool for replaying traffic statefully is *TCPopera*[17] which aims to maximize the preciseness of TCP traffic replay. It has a mixed analytic model-based and trace-based approach. The timing and packet properties are processed into a model, where as the traces contents is replayed as is.

This tool has the following objectives: no ghost packet generation, different traffic models support, inter-connection dependency, scalability and extensibility. The last objective means that TCPopera should have a possibility to be deployed in large test environments, with different models and protocol implementations, environment transformations and provide a possibility to install the tool to any arbitrary network.

TCPopera has two phases: preprocessing traces and replaying traffic. During preprocessing the tool extracts traffic parameters, network configuration and data flows. A user can then change those. The main components of *TCPopera* are Flow Preprocessing, IP Flow Preprocessing, TCP Timer, TCP Control, Packet Injection, Packet Capturing.

Inter-connection dependencies are supported only within one IP flow. It is done by keeping the order of packets for communication between two hosts.

Packets are sent via the Packet Injection component through TCPopera nodes to a particular virtual address.

2.8.5 Comparison. Feature mapping

Table 7.1 contains comparison for the described tools for a list of features.

3

Network traffic analysis

Replaying traffic is not as straightforward as simply sending packets from original trace because of dynamics and difference in network devices. This can affect different protocols in different ways and requires to be considered for the tool. Another situation to consider is that traffic undetermined dynamics not only affects certain packets directly, but also cascades to parameters of dependent traffic.

3.1 Protocol sessions

IoT traffic can be represented by various protocols including but not limited to DHCP, ARP, IP, ICMP, TCP, UDP [18]. A trace related to a protocol session can be characterized by

- the participating hosts
- the protocol header information
- the packet order
- the packet contents: the data itself and how it is distributed over packets
- the packet timing

Depending on a test-bench goals, each of these characteristics can cause challenges for a replay tool. For example, for network load testing, the timing and contents size are important; for deep packet inspection security devices, packet contents and order are important. In our case, the hosts turned out to be the most important feature to identify the acting device.

Table 3.2 represents some of the remote hosts and protocols in the traffic produced by Arlo Camera. As can be seen from the table, TCP and UDP are mainly used as the transport protocols.

remote host address	protocol	additional info
broadcast	DCHP	Over UDP
broadcast	ARP	
gateway	ICMP	echo (type 8 and 0)
gateway	DNS	Over UDP, port 53, for time-a.netgear.com
193.166.4.60	NTP	Over UDP, port 123
23.67.133.117	HTTP	Over TCP, port 80
52.31.233.143	HTTPS	Over TCP, port 443
ff02::16	ICMPv6	echo (type 8 and 0)
broadcast	ICMPv6	echo (type 8 and 0)
209.249.181.91	NTP	over UDP

Table 3.2: traffic example started by the IoT device

3.1.1 Hosts

The participating hosts are always important traffic features, because the network at least needs to know where to send the packets. IP packets need to have a source and a destination IP address [19]. Ethernet headers of packets need to have a source and a destination MAC address.

One of the tricky problems with replaying traffic is making the simulating instance MAC and IP addresses correspond to the ones used by the simulated device in the replayed trace. Obviously, there are two ways of doing it: setting the addresses of the simulating instances to match those in the trace, or changing the addresses in the replayed trace addresses to match those given to the simulating instances.

For testing an intermediate device, the addresses of this device should also be taken into account along with other hosts working within the test-bench.

A difficult technical problem arises when the IP address of one of these nodes is unknown in advance and allocated dynamically during the test. These cases will be considered later in the section 3.2.

3.1.2 TCP and UDP ports

Ports are the basic feature distinguishing sessions for these two transport protocols. A pair of IP address and a pair of TCP or UDP ports defines a session. When replaying traffic, a testbed must maintain these parameters coherently to that the sessions from the original trace and retained.

However, the opacity of the intermediate device (router under test (RUT)) violates the original trace by changing this pair. For instance, a router with DHCP feature and NAT can change both the LAN side ports and addresses! So, when simulating TCP and UDP connections over such a device, it is important to change the destination port and address for traffic going towards a host in the local network to the one dynamically given by gateway.

One potential way to do that is to know the ports in advance and change ports in the replayed traces, as it is done with Tcpreplay and Tcprewrite [15][20]. Another way is to do the mapping on the fly, after getting a response from RUT. The first case is impossible unless the router cooperates in such a way that the dynamic port is predicted correctly. In the second case, the replaying part is supposed to be more intelligent: remember the port given by the RUT and substitute the original port from the trace by it.

3.1.3 TCP

TCP takes a significant part of most IoT traces [18]. This protocol is special for its handling of lost packets, which are replayed. As a result, a trace can contain duplicate packets, packets with a messed order, or missing packets.

A TCP trace may be of two types: client-server communication, where a client sends requests and gets responses, which is easier to reproduce in a plausible way, or two-directional independent flows that is difficult to be divided into messages. The testbed should be able to refragment TCP flows, but it must not confuse the logical order of requests and responses. A client-server communication could be identified by the application-layer protocols that are architecturally designed to have request-response structure, such as HTTP usually operating at port 80. Another potential way is to check how data is sent within ACK packets. For instance, when a client does an http request it has some payload with request data. The server always replies with an empty ACK packet first to inform that request is received and then processes it and sends a reply. Whereas, in two directional flow, when the load is enhanced, in order to save network from numerous empty ACK packets, a "piggibacking" technique is used. In this scenario, a communication participant delays answering ACK packet, waits for data to send from its side and if there is, attaches the acknowledgment to the outgoing data frame.

When replaying TCP packets, the order and content distribution over packets is important to test IDS because, in order to go through IDS unnoticed, an adversary can do the following tricks. One trick is to fragment the data flow into small pieces, so that each piece does not contain any malicious data by itself, causing an IDS that does not have flow reassembly to ignore those packets. For those IDS that do reassembly flow, an adversary can rearrange the packet order in such a way that there would be two or more parts, each of which can not be detected as malicious. The IDS needs to process and assemble other flows, and at some point, it might drop the flow started by the attacker. The attacker can wait for a certain amount of time for the IDS to flush the buffer and then send the next batch of his flow. This also implies the requirement for the testbed to have precise packet timing when replaying flows and the ability to add delays.

IP packet fragmentation can also become a problem when replaying TCP. Adversaries can use this phenomenon to hide attacks, making session more difficult to reconstruct for IDS.

In comparison with UDP, manipulating TCP is complicated. TCP includes the following fields: sequence number, acknowledgment number, multiple flags, window size, data offset and urgent pointer. Those fields are dependent on each other and on previous information sent by the communicating host. Violation of TCP packet structure can cause RUT to abolish a session and fail a test purpose. More information can be found in the book [21].

3.1.4 UDP

The User datagram protocol (UDP) is significantly simpler than TCP. Like the previous protocol, it has source and destination port, payload length and checksum — these are the only fields that require consideration during replay.

Due to the simplicity of the protocol and absence of any inter-packet dependencies, it is very straightforward to replay. However, as UDP does not have any sequence or acknowledgment number, it is uncertain if the application layer meant the packets to go in the particular order captured in a given trace.

3.1.5 ICMP

The Internet Control Message Protocol (ICMP) plays an auxiliary role. It has dozens functions, defined by the *type* and *code* fields, many of which are deprecated by now. ICMP does not have ports, and when packets go through NAT, router maps incoming packets to LAN hosts differently in the way described below.

The most well known example is the echo function triggered by the ping utility. It is used to check accessibility of a particular host via the IP network. Often, the purpose of using it is just checking an internet connection. Field *type* is set to 8 for echo request and to 0 for echo response messages. The ICMP *ID* field is used to map the incoming packet to the original LAN host and it also defines the session along with the participating hosts (similar to host-port pairs in TCP). The *sequence number* field is used to identify a packet in case of packet duplication or loss, which can be seen by skipping a sequential number. The field *data* is used for a message that needs to be echoed. Potentially, any of the last two fields could be used by an adversary to slip past IPS and transfer some data.

Another important function is “Parameter Problem: Bad IP header” marked with ICMP type 12. NAT identifies the source host by parsing the header information from the ICMP packet payload.

ICMP type 3 is used to notify the sender of a packet that the destination address is unreachable. Taking this into account could be used to terminate unfinished TCP or UDP sessions, thus optimising test the running time. In any case, the RUT may send such error message at times that differ from the original trace, and it may react to ICMP messages differently.

3.1.6 Time to live

IP protocol has *time to live* (TTL) field. It should also be considered, as this field can cause router to stop the flow from reaching the destination. This is actually a hop limit meaning the number of times the packet can be transferred from one router to another, preventing the packet from circulating indefinitely. The value of this field is decremented every time the packet is forwarded. If a router receives a packet with TTL equal to zero, it drops the packet (fig. 3.1). Omitting the original value this field may lead a replay test to produce incorrect results.

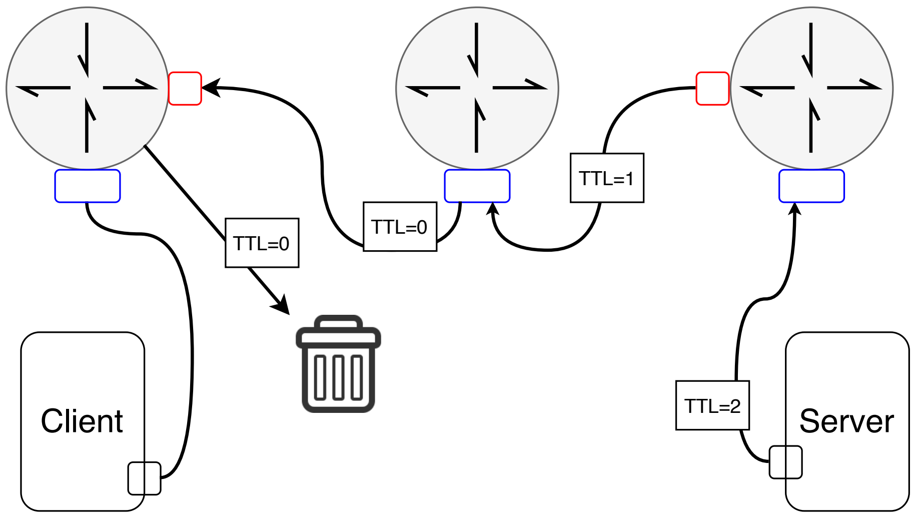


Figure 3.1: Time to live

3.1.7 Checksum and length

The IP and transport layers have checksum and length fields that protect packets from errors. Since packets might be impossible to replay as is in the original trace, it is important to keep these fields correct to avoid packets from being dropped or processed in a wrong way.

3.2 Inter-protocol dependencies

In computer network, it is common for contents of one session to depend on contents of another. These dependencies can potentially be found in application layer, but it is not so easy to find out by looking at bare traces and hardly possible to enlist all such cases. What is more important to focus on is the network layer dependencies. Host addresses is the data most frequently affected by depended protocols and session.

3.2.1 DHCP

Dynamic host configuration protocol (DHCP) is used to provide an IP address to a host. Normally, the DHCP session happens when the host first connects to the access network, and all the later IP based sessions use the acquired IP address as source. However, it should be kept in mind, that this is not always the case — nothing prevents a host from changing its IP address.

3.2.2 DNS

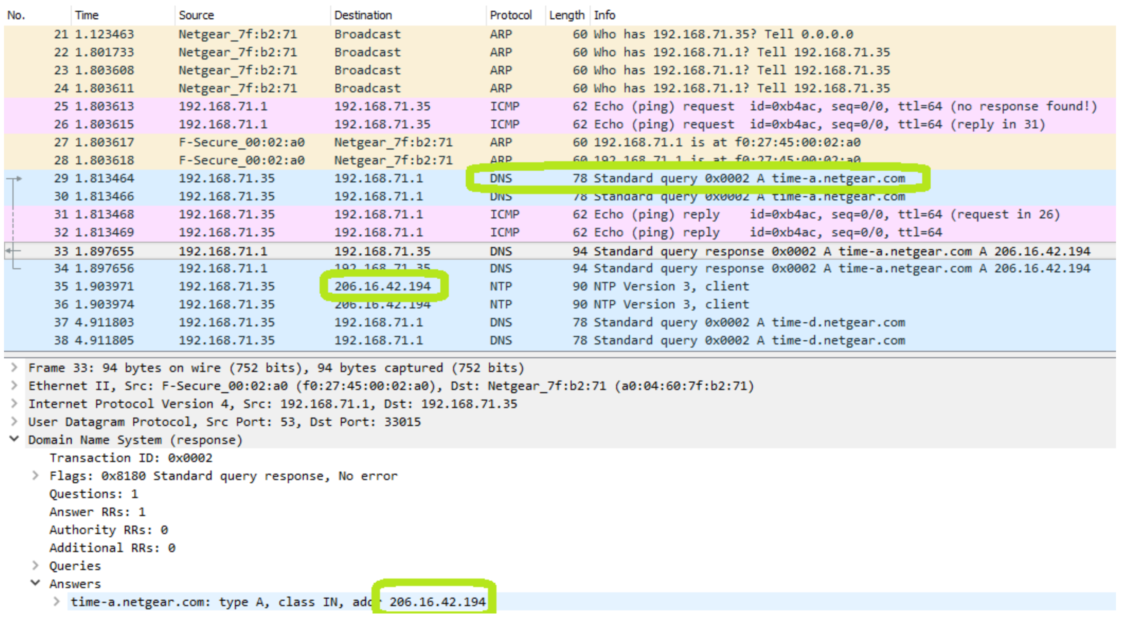
A major technical problem is changing the simulated servers' IP according to DNS responses from gateway.

There are two ways of dealing with this problem: reading the responses from DNS and changing IP addresses accordingly; or emulating DNS servers to provide the IP address from the trace or the one that we want actually to see. Both cases are not so simple to handle.

For both cases, it needs to be determined that particular sessions are dependent on DNS request. The figure 3.2 shows an example of (most probably) related sessions from a trace captured from a real IoT device (Arlo camera). As can be seen at packet number 29, the IoT device is requesting the NTP server's IP address through DNS request. Router replies with packet 33, and this address is used in packet 35.

In the provided example, the NTP session follows the DNS request uninterrupted. However, we can assume that there may be some unrelated packets in between. Based on this and many similar examples, we can derive an algorithm for searching for interconnected DNS sessions. For each DNS session, the IP address is extracted from the response. This address is searched for in the following sessions as the destination of packets sent by the IoT device. The first found session is linked to the latest corresponding DNS response. Caching the DNS results by the IoT device is another situation to consider, meaning that the same IP address is used for multiple sessions without making a DNS request again. We might want to apply the DNS dependency to all server addresses in the trace after the DNS request.

For the first DNS solution number one — DNS monitoring, the IoT simulating device needs to send DNS request to RUT and extract the IP address from the response. Then, this



```
No.   Time       Source              Destination         Protocol  Length Info
   21 1.123463   Netgear_7f:b2:71    Broadcast           ARP       60 Who has 192.168.71.35? Tell 0.0.0.0
   22 1.801733   Netgear_7f:b2:71    Broadcast           ARP       60 Who has 192.168.71.1? Tell 192.168.71.35
   23 1.803608   Netgear_7f:b2:71    Broadcast           ARP       60 Who has 192.168.71.1? Tell 192.168.71.35
   24 1.803611   Netgear_7f:b2:71    Broadcast           ARP       60 Who has 192.168.71.1? Tell 192.168.71.35
   25 1.803613   192.168.71.1        192.168.71.35       ICMP      62 Echo (ping) request  id=0xb4ac, seq=0/0, ttl=64 (no response found!)
   26 1.803615   192.168.71.1        192.168.71.35       ICMP      62 Echo (ping) request  id=0xb4ac, seq=0/0, ttl=64 (reply in 31)
   27 1.803617   F-Secure_00:02:a0   Netgear_7f:b2:71    ARP       60 192.168.71.1 is at f0:27:45:00:02:a0
   28 1.803618   F-Secure_00:02:a0   Netgear_7f:b2:71    ARP       60 192.168.71.1 is at f0:27:45:00:02:a0
   29 1.813464   192.168.71.35       192.168.71.1        DNS       78 Standard query 0x0002 A time-a.netgear.com
   30 1.813466   192.168.71.35       192.168.71.1        DNS       78 Standard query 0x0002 A time-a.netgear.com
   31 1.813468   192.168.71.35       192.168.71.1        ICMP      62 Echo (ping) reply    id=0xb4ac, seq=0/0, ttl=64 (request in 26)
   32 1.813469   192.168.71.35       192.168.71.1        ICMP      62 Echo (ping) reply    id=0xb4ac, seq=0/0, ttl=64
   33 1.897655   192.168.71.1        192.168.71.35       DNS       94 Standard query response 0x0002 A time-a.netgear.com A 206.16.42.194
   34 1.897656   192.168.71.1        192.168.71.35       DNS       94 Standard query response 0x0002 A time-a.netgear.com A 206.16.42.194
   35 1.903971   192.168.71.35       206.16.42.194       NTP       90 NTP Version 3, client
   36 1.903974   192.168.71.35       206.16.42.194       NTP       90 NTP Version 3, client
   37 4.911803   192.168.71.35       192.168.71.1        DNS       78 Standard query 0x0002 A time-d.netgear.com
   38 4.911805   192.168.71.35       192.168.71.1        DNS       78 Standard query 0x0002 A time-d.netgear.com

> Frame 33: 94 bytes on wire (752 bits), 94 bytes captured (752 bits)
> Ethernet II, Src: F-Secure_00:02:a0 (f0:27:45:00:02:a0), Dst: Netgear_7f:b2:71 (a0:04:60:7f:b2:71)
> Internet Protocol Version 4, Src: 192.168.71.1, Dst: 192.168.71.35
> User Datagram Protocol, Src Port: 53, Dst Port: 33015
v Domain Name System (response)
    Transaction ID: 0x0002
  > Flags: 0x8180 Standard query response, No error
    Questions: 1
    Answer RRs: 1
    Authority RRs: 0
    Additional RRs: 0
  > Queries
  v Answers
    > time-a.netgear.com: type A, class IN, adc 206.16.42.194
```

Figure 3.2: Layer subsection schema

IP address must be used in the following session linked to the DNS session. The crucial point here is not to forget to use the same IP address on the server side.

In this solution, the complexity is in DNS replay logic and other service-discovery communication (skipping the RUT) to set up the correct IP addresses. This has a significant impact on the testbed architecture.

The DNS mocker solution takes all complexity away from the actual replay tool. The component emulating the DNS server is nicely set up separately, and its task is to extract the IP addresses wanted for the replayed trace and to inject them to the DNS responses. However, this solution cannot cover the RUT cached DNS results case, potentially causing the traffic replay test to fail.

One important note is that, in traces observed during the described project, no DNS requests going from the tested router were found. This means that the router itself can cache domain data for very long far exceeding several tests period of time, and work as the DNS server by itself. This fact excludes the DNS mocking solution.

3.2.3 ICMP relations

A TCP replay process could be affected by ICMP errors [22]. Errors can be classified as soft and hard. Soft errors, like transient network failure, should be handled, whereas hard errors should cause a session to terminate. There are two issues to consider in replaying errors: reacting from client side, and sending an error from WAN side. This might require a complicated manipulation.

For example, during an ongoing TCP session, a router in the middle suddenly cannot find the destination and sends an ICMP packet with host unreachable message, causing the client abort the session. First of all, it is important to keep the order of the packets, when replaying. Another problem is to link the TCP session with ICMP message in the saved trace.

№	Layer	Protocol Example
1	Link	Ethernet
2	Network	IP
3	Transport	TCP
4	Application	HTTP

Table 3.3: Network layers example

The ICMP packet needs to be parsed to find out what headers it refers to. If hosts' addresses in the replay are different from the original trace, they need to be changed accordingly in the ICMP payload as well.

Another problem to consider is keeping the replay close to reality when replaying using TCP socket connection. One might not link ICMP and TCP session and terminate the latter with FIN and FIN ACK packets, what would not be correct in case server connection lost.

3.2.4 Secure DNS

DNSSEC is the protocol that allows DNS to be authenticated (digitally signed) but does not provide confidentiality. This means, that the IP address for the next session should be extracted like for a DNS session. The problem with DNSSEC is that the timestamps in the signed records in the trace may have expired. This can cause the RUT to block the expired resources during the replay.

Another case to consider is the DNS over HTTPS. This technology helps to keep DNS requests confidential. Unless session is decrypted, there is no way to know exactly what domain name was requested and what IP address was transmitted. But actually, we don't really need to know it and just replay traffic as is with the original remote IP address, because the RUT does not know the correct IP either. Indeed, DNS over https is designed to bypass the local DNS server and cahce in the local router. Thus it creates no particular problems in the replay. Besides, there is hardly a motivation for an IoT to hide its DNS requests, unless it is hacked.

3.3 Network layers

Open Systems Interconnection model (OSI) defines a stack of protocols to be used for a packet transmission. Each protocol puts its load on top of the previous one. This stack can be represented as layers (see table 3.3).

Traffic can be replayed at a particular network layer, meaning that the corresponding software would be used to build the packet on top of it.

If the lower layer is chosen to replay traffic at, the developer probably needs to take care of all the upper layers. The point is that the packet will go through fine at this particular segment but can be dropped or misinterpreted later. For instance, we have a setup of a router, host A in the router's LAN and another host B connected to the routers WAN interface (any of them may be virtual) illustrated on figure 3.3. The router has a NAT hiding IP address of host A. The task is to replay a TCP flow between hosts A and B initiated by host A. Since it is a test over router reaction, the IP address of host B is determined by the original trace, whereas IP address of host A can be dynamically defined.

If we try to replay it at the link layer, then we need to adjust the MAC addresses of all packets from the original trace to the addresses used by the replaying instances. For a packet sent by host A, we set the source MAC address of host A and destination address to routers MAC address. Then, we copy the upper layer payload as is from the original trace. It will work perfectly fine for the link host A – router. But when it comes to forwarding the packet from the router further to host B, it might turn out that the IP address of host A is different in reality from the one saved in the trace. So, we have to modify the source IP address of the packet to the one dynamically given. Then the packet (transport layer part) will reach host B.

Now, when we replay the responding packet from host B, the source and destination MAC addresses are set to host B and the router respectively. The source IP address should be the same as in the trace, but the destination IP address should be set to the router's one (marked as "IP 3" on figure 3.3). But this is not enough, the second packet can be dropped on router because it has the wrong destination TCP port — different from the one assigned by the NAT which does not correspond to the one in original trace.

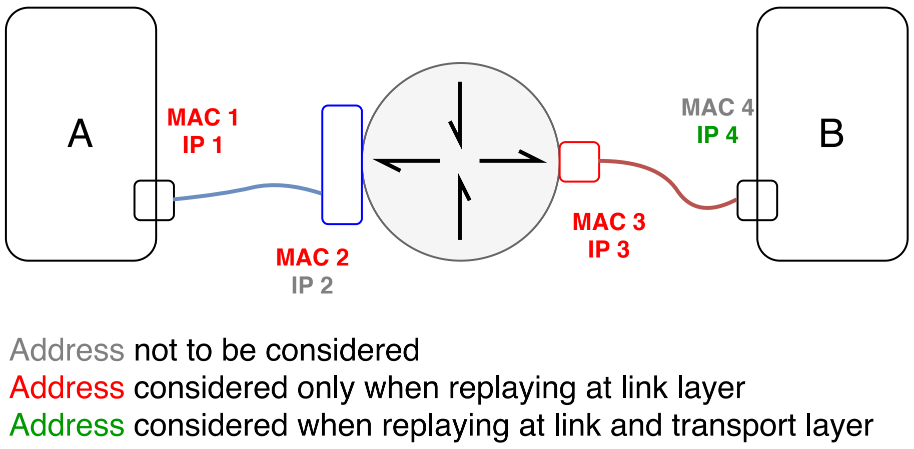

A
MAC 1
IP 1
MAC 2
IP 2
MAC 3
IP 3
MAC 4
IP 4
B
Address not to be considered
Address considered only when replaying at link layer
Address considered when replaying at link and transport layer

Figure 3.3: Layer subsection schema

If the same traffic is replayed at the transport layer, the only thing we need to think about is the destination IP address of packets heading to host B. And we still need to parse the packet to find out the port that host B listens to. But those are defined during a setup — no parsing or packet payload modification is needed during the run-time. The rest is done by the native software or hardware of the hosts.

The higher layer is used, the easier it is to replay traffic reliably, as shown above, but the less possibilities we have to imitate the original traffic. For example, if replaying at application the HTTP layer, it might not be possible to change the timing of the packets properly; if replaying the same traffic at the transport layer (TCP), it is not possible to change or maintain original order of packets for the purpose described in 3.1.3.

3.4 Session order and dependencies

The major cases are considered in 3.2. This section will present an abstract way to think about order of packets and synchronization of sessions.

The protocol, for which replay implemented during this work (TCP, UDP, ICMP echo) have sessions. For TCP and UDP, a session is defined by the host-port pairs. ICMP echo sessions are defined by the identifier field. We cannot expect that every protocol has sessions. For instance, ICMP Destination Unreachable [23] is a single packet sent from a router. In the theory developed in this work such single packets along with any packet that cannot be identified as part of another session due to any reasons will be considered as a separate session.

So, let us assume that we have an arbitrary tool that can process an original trace of recorded packets sequence $T = (r_1, r_2, \dots r_n)$, where r_i is i-th packet in the trace, into a sequence of $P = (p_{1,1} \dots p_{k,m} \dots)$. For a packet $p_{k,m}$, k is a session number and m is its sequential number in that session. So a session is defined as $s_i = (p_{i,1}, \dots, p_{i,k_i})$. And as a result we have a set of sessions $S = \{s_1, \dots s_m\}$.

Each packet $p_{i,j}$ from S has a corresponding original packet in original trace T defined by the function $O : P \to T$. Let us define a packet r_i is ***earlier*** than packet r_j if it appears earlier in the trace T and denote the comparison as $r_i < r_j \Leftrightarrow i < j$. We can also expand this designation to the packets in the processed *trace set P* as $p_{i,j} < p_{k,l} \Leftrightarrow O(p_{i,j}) < O(p_{k,l})$; and to the *session set S* so that one session is *earlier* than another if the its first packet is *earlier*: $s_i < s_j \Leftrightarrow p_{i,1} < p_{j,1}$.

Let us define a session s_i ***aftercomes*** s_j if s_i starts after s_j terminates and denote: $s_i \gg s_j \Leftrightarrow \forall k, l : \ s_{j,k} < s_{i,l}$.

Simplifying the definitions above for sessions interdependencies would be useful to understand the session picking algorithms. Expression $s_1 < s_2$ means that starting s_2 requires s_1 to be started. Expression $s_2 \gg s_1$ means that starting s_2 requires s_1 to be terminated.

Let's acknowledge three very simple facts that will help us to optimise session picking algorithm.

Statement 1: The relation *earlier* is transitive.

Proof. Let $s_1 < s_2$ and $s_2 < s_3$. $\Rightarrow s_{1,1} < s_{2,1}$ and $s_{2,1} < s_{3,1} \Rightarrow O(s_{1,1}) < O(s_{2,1})$ and $O(s_{2,1}) < O(s_{3,1})$. Since those comparisons are deducted to integers that are known to be transitive on comparison $\Rightarrow O(s_{1,1}) < O(s_{3,1}) \Rightarrow s_1 < s_3$. ■

Statement 2: The relation *aftercomes* is transitive.

Proof. Let's denote the last packets for s_1, s_2 as $s_{1,m}$ and $s_{2,n}$ respectively. Let $s_2 \gg s_1$ and $s_3 \gg s_2$. $\Rightarrow s_{3,1}\ s_{1,m} < s_{2,1}$ and $s_{2,n} < s_{3,1}$. Since $s_{2,1} < s_{2,n}$, $s_{1,m} < s_{3,1} \Leftrightarrow s_3 \gg s_1$. ■

Statement 3: If one session *aftercomes* another, then the latter one is *earlier* than the first session.

Proof. Let $s_1 \gg s_2$ and $s_{2,k}$ be the last packet of s_2 then, $s_{1,1} > s_{2,k} \Rightarrow s_{1,1} > s_{2,1} \Leftrightarrow s_2 < s_1$. ■

During this work replay principles were worked out for sessions replay.

3.4.1 One current replay flow

If the tool is running no more than one session at a time, then the most straightforward way to replay the trace would be to sort sessions by the start time ascending. So, the order would be $s_1, s_2, \ldots, s_m$, where $s_i < s_{i+1}$. This states the first rule of replay.

For this case, the first rule is enough to keep the order of interdependent sessions.

3.4.2 Concurrent replay flow

In reality, a device can have multiple sessions running at the same time, and it can be beneficial to be able to simulate this. The first idea would be the same as above: a session should start before the previous one does if $s_i < s_{i+1}$.

The second replay rule is that session, if s_i *aftercomes* s_j, s_i cannot be started until s_j is not terminated.

3.4.3 Concurrent implementation algorithm

The first rule implementation guarantees that sessions start in the same order. The second rule guarantees that TCP will not start before DNS, or ICMP error is not sent before the TCP packet that caused it.

However, concurrent implementation is not as simple as the rules it would follow. This section offers possible algorithms for implementing them.

Separating functionality is usually a good idea to improve abstraction and maintainability. This paper will introduce two algorithms. One is for processing a session set (example in fig. 3.4) into graph structure (fig. 3.5). The other is for implementing the pipe giving out the sessions. The first algorithm has strict assumptions on trace sessions dependencies, which is not very likely to happen in real-life devices. That is why it is better to keep the processing and the sessions piping functions separated. In this work, only two events are considered: session start and session end. This is also not covering all cases possible. Both algorithms could be extended with more session dependency types.

Both algorithms share denotation of sessions set S, dependencies graph $G = (S, E)$, where the set $E \subset S \times S \times T$ represents the dependency edges from one session to another of type $t \in T$ The type is either start requirement or termination requirement.

3.4.4 Dependencies build

For this algorithm, we will use two types of dependencies $T = \{b, t\}$ — b for "begin", so that $(s, q, b) \in E$ means that $q \in S$ requires $s \in S$ to start first; and t for "terminate", so that $(s, q, t) \in E$ means that $q \in S$ requires $s \in S$ to terminate first.

3.4.5 Dependency fulfilling

The idea is to unlock sessions for replay on certain events after their dependencies are satisfied. An offered technical implementation is to remove the edges from the dependency graph. Let us consider an example where $s_3 \gg s_2$ and $s_5 < s_2$. Termination of replaying s_2

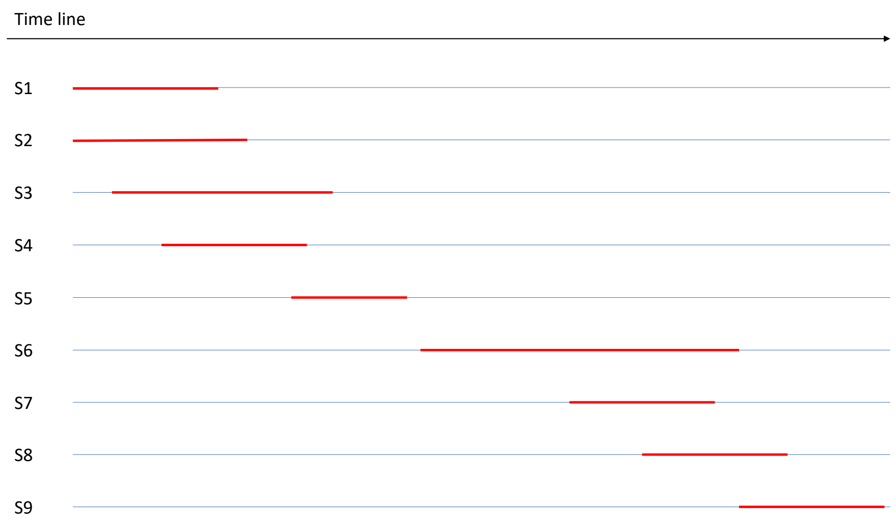


Figure 3.4: Sessions lifetime example

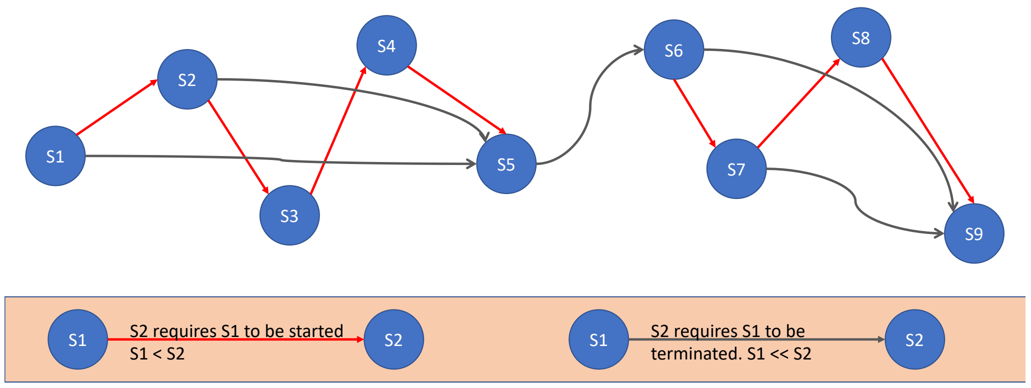

S1
S2
S3
S4
S5
S6
S7
S8
S9
S2 requires S1 to be started
S1 < S2
S2 requires S1 to be terminated. S1 << S2

Figure 3.5: Sessions requirements

Algorithm 1 Dependency Graph Building Algorithm

```
$E \leftarrow \{\}$
**for** $s \in S$ **do**
    **for** $q \in S$ **do**
        **if** $s \gg q$ **then**
            add $(s, q, t)$ to $E$
        **else**
            **if** $q < s$ **then**
                add $(s, q, b)$ to $E$
            **end if**
        **end if**
    **end for**
**end for**
```

should fire an event to remove the dependencies of s_3 and s_5 on s_3. The dependency removal should also trigger the session pipe update.

The dependency removing function is pretty simple but is difficult to for a proper pseudo-code description, and it has been omitted here.

3.4.6 Session pipe algorithm

The session pipe is a class that receives events of removing dependencies from the constructed graph as those are satisfied and provides input to a queue for the new sessions to start.

Piping the sessions is not a straightforward technical task. There should be a queue between the session-start giving process and the processes replaying sessions. Not every request for a session can be supplied due to dependency conflicts. Events triggering dependency removal can also trigger a new session ready to be started added, which has been added to the queue.

Algorithm 2 Session Get Pipe Function

```
function PushSessionsToQueue(set $E$, queue SessionQueue)
    for $s \in S$ do
        if $E$ does not contain $(x, s, y)$ then
            SessionQueue.push($s$)
        end if
    end for
end function
```

4

Testbench overview

The schema of a home network with a security router is shown in figure 4.1. The network contains one secure router as a gateway and other devices connected to it directly. The goal of this work is to simulate an arbitrary device from this network.

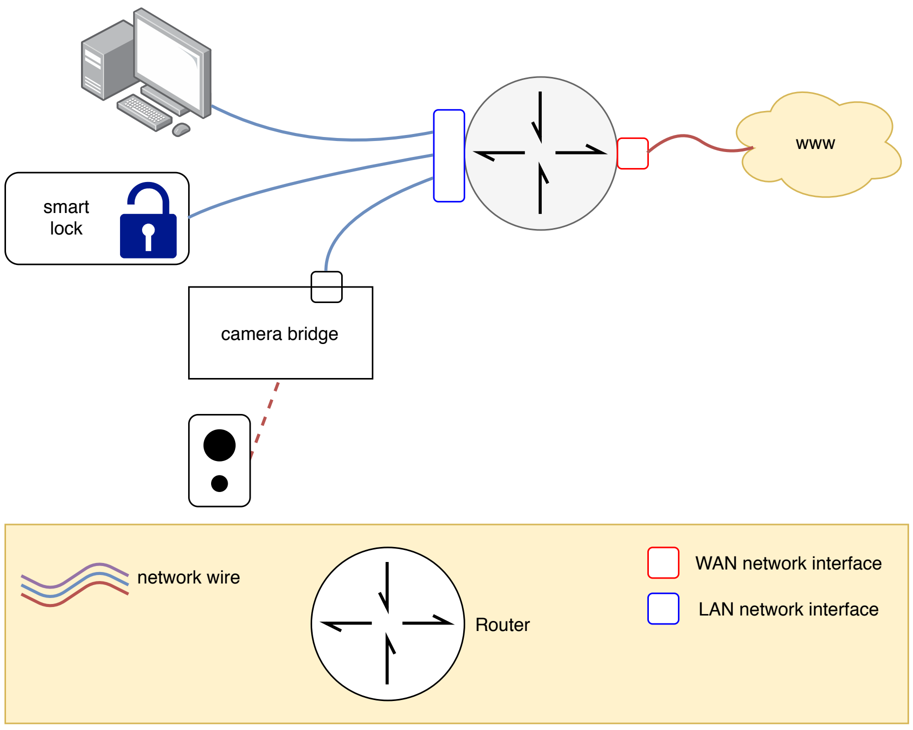


Figure 4.1: Security router in a home network

The purpose of the testbed is end to end testing of a router, thus requiring its physical presence. Hence, the workspace is a testbench with multiple devices on it operated by a human through an operating device (OD), which is a personal computer. The testbench also includes a switch with a mirroring port and a gateway router that connects the whole system with the Internet.

4.1 Workflow

The testbed operation is separated into three independent phases: traffic capturing, traces processing and replaying them (fig. 4.2). Every phase produces an input for the next one.

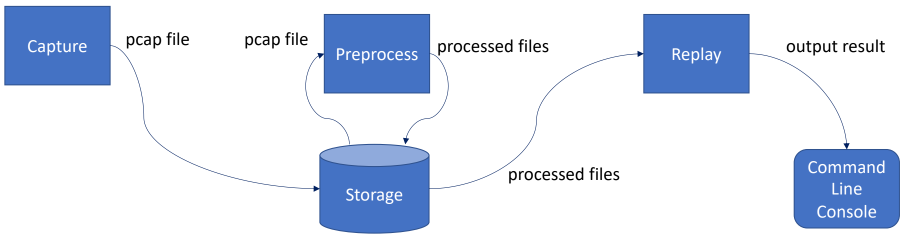


Figure 4.2: Testbed processes

Separating the processes allows independent development, execution and optimisation of each phase. The only requirement is the format of the “pipe” parameters. However, the testbench was designed in such a way that every phase can be done within the lab with minimal changes to it when moving to the next step.

4.1.1 Capturing

First step to reach the set goals would be to capture actual traffic produced or triggered by a LAN IoT device and to save it for replaying later. An operating person should set testbench into the state shown in figure 4.3, make sure that RUT is up and running, prepare the IoT device at the testbench, connect the ethernet cable with from the mirroring port to the OD, launch the capturing process and then start the IoT device. The expected result is a pcap file saved on an OD. This file may contain some side effect or unrelated packets — those can be filtered in next phase.

The process is launched in an OD terminal by a command that takes the following parameters: output path, defining the destination file; network interface to listen from; capturing timeout time, unset by default and working until the user interrupts the process; traffic filter as optional parameter.

However, it is not necessary to use the developed testbench for getting network traces. Any pcap files captured or created outside the lab can be used for the later phases.

4.1.2 Preprocessing

This process transforms given pcap file into files that can be used by the replaytool. The resulting files are specific to the replay tool. They include data integrating the overall picture like session order and parameters, and each session separately. The session format varies and depends on the implementation of a particular protocol replaying component.

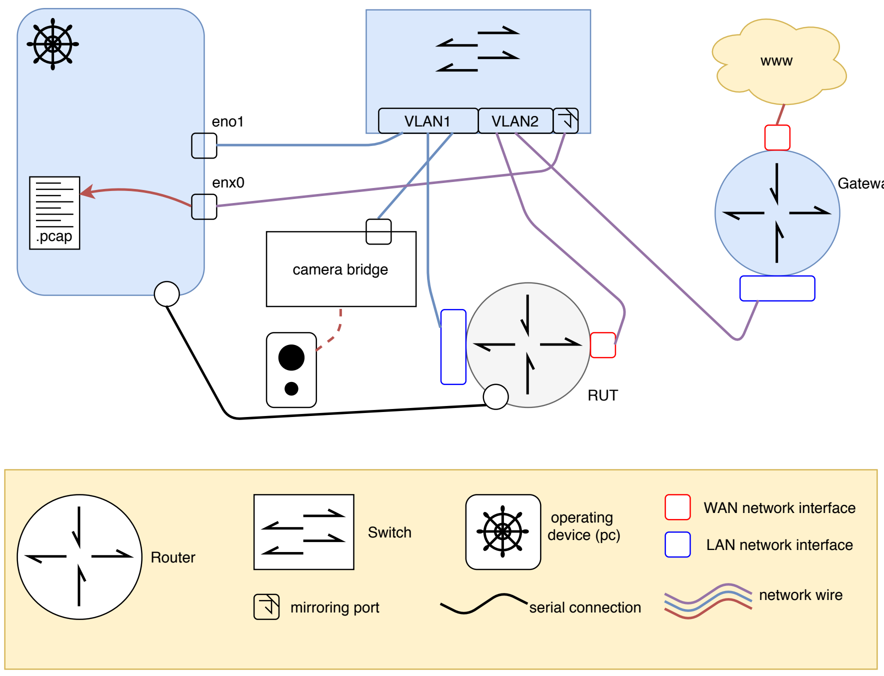


Figure 4.3: Capture Traffic Scheme

This phase is divided into two steps. The first one is the transformation of pcap file into the miscellaneous files prepared for replay tool. This is done by a command line script called *preprocess*.

The resulting output can contain extra traffic that is not needed for the upcoming test. Another step thus is required to refine the data. A utility program was developed named adjust. It processes the output json files into similar ones but filtering unnecessary sessions and changing the IP addresses to a the desired ones. The resulting json files are named in same manner but using *"input"* prefix by default, meaning they are ready to be taken as input by the replaying tool.

Session parameters like the participating hosts, ports or session order might have a need to be modified for the test purpose. After a second step of refinement, the data is filtered and it should be easy enough to find to modify the needed session parameters.

This phase is usually done on OD, but it is not a requirement. It can be run on any computer with Linux OS outside the lab.

4.1.3 Replaying

Before starting the replay process, the operator should make sure that the network set up is correct and the preprocessed set of files information corresponds to it. For instance, the gateway IP address in the source files should match the RUT IP address.

The replaying phase requires the operator to set the testbench in the state shown on figure 4.4 by connecting network wires. The replay process is then started by a command line utility. The input is a main json file in the same directory as the set of preprocessed files and the simulated device MAC address.

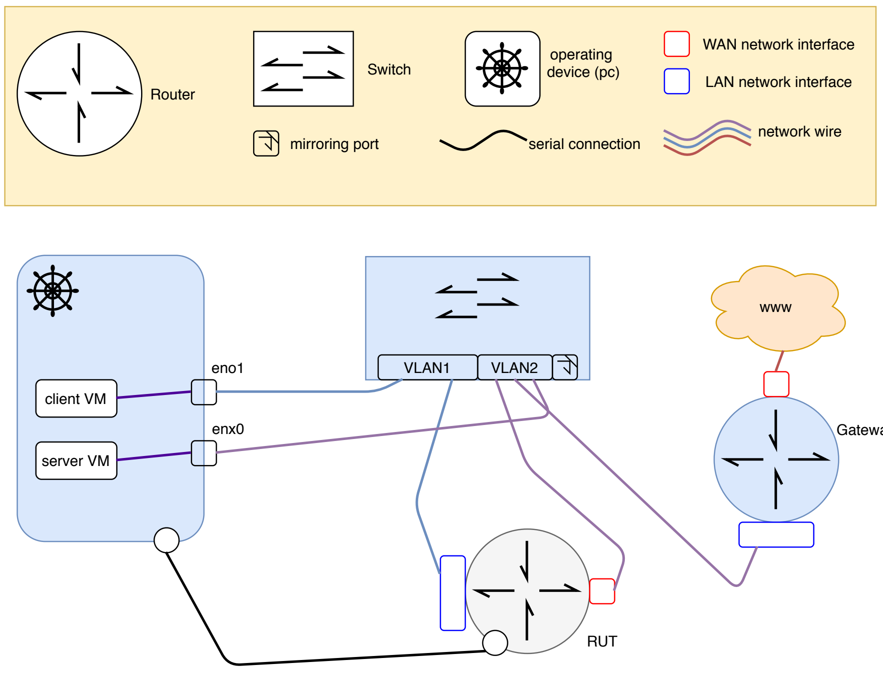


Figure 4.4: Replaying Traffic Scheme

MAC address can be input directly as a hexadecimal code, or autogenerated by the used tools. An utility was also created to give pass a device name as a parameter instead of MAC address. It is handy if experiment is composite: at first you want RUT to identify you simulated device and then reuse it. For instance, the first time you run the test with the original trace. Then you manually modify the input by adding an extra session to simulate an anomaly and rerun experiment with the same device name corresponding to its original MAC address.

All the information about the process is output to the terminal. In the end, a RUT specific utility checks for anomalies detected.

5

Upper level implementation

It was decided that a general purpose computer is used to control all the process. It will be referred to as the operating device (OD). Ubuntu 16.04 was chosen as the operation system for the OD because Linux-based operating systems allow easier manipulation of network settings and SSH access. Another device in the testbench is the router under test (RUT) itself. Other devices installed to the testbench will be described in the following sections as the need for them is introduced.

Python 3.0 was chosen as the main development language because it is very simple, suits prototyping development, is runnable on all major operation systems and has a great number of libraries including those working with network packets.

5.1 Traffic capturing setup

5.1.1 What traffic needs to be captured

We are interested in traffic flowing through the RUT. It contains incoming and outgoing packets from both sides of the RUT: LAN and WAN communications between an IoT device and a server.

Mostly flows just go through the router, but it should be kept in mind that the router has a probability to drop packages going from both the LAN and WAN interfaces, or to block the connection entirely. Hence it would be beneficial to monitor both sides of the RUT in te network.

5.1.2 Way of capturing

In this work, a mirroring switch HP 1820 8G was used to obtain traffic from the real devices. This switch has one mirroring port where it duplicates all packets incoming to any other ports. In order to capture the traffic, we set up an Ethernet connection from that port to the OD and started a recording utility listening to the network interface. This way, traffic can be captured by any device, which relieves the system from strict dependencies of specific recording software.

In order to capture traffic on both sides of the RUT, HP 1820 8G was configured to have four of the ports on the LAN side aggregated to *VLAN1* and three on the WAN side aggregated to *VLAN2*. The eighth one is the mirroring port. As shown in figure 4.3, every device that has its traffic captured is connected to VLAN1 of intermediate switch before the router. The RUT LAN interface is connected to VLAN1 also. The RUT WAN interface is connected to the switch's VLAN2 along with the whole-testbench network gateway. The mirroring switch is connected to the OD to record the trace. The switch is set up manually once and does not require manipulation except changing the wired connection depending on the testbench aims.

Another issue to consider is the speed of the traffic. The HP 1820 8G Ethernet ports support 100 Mbps and 1000 Mbps speed. Hence, to be able to reflect all traffic to the mirroring port, the total traffic speed on all other ports must not exceed the speed on the mirroring one. To provide this condition, mirroring port speed was set to 1000 Mbps while the others ports to 100 Mbps.

5.2 Replay Setup

The idea of the replay setup is to have a simulated IoT device (client) and simulated remote hosts (servers) under control of the OD to run their communication through RUT. It was decided that both client and server instances will be running on the OD.

The problem is that the servers IP address flowing through the RUT must be the same as in the original trace, meaning that RUT will send the packets into the outer internet as it would do in a typical setup. In order to take control of the simulated traffic, a controllable gateway router was introduced to the testbench. The one that was used had an the OpenWRT operating system. The replaying program sets its IP routes to redirect the server oriented traffic to the OD interface to which the server simulating instances are connected, without the RUT knowing anything about it. The connection setup is shown in figure 4.4.

In the developed testbench, for the trace replaying, the OD requires two wired connections to the intermediate switch: one connected to VLAN1 and another to VLAN2. The LAN packets from the trace are to be played through VLAN1 connection and WAN packets through VLAN2. The schema is shown on figure 4.4. Naturally, the client simulating instance is connected to VLAN1 and the server simulating instances are connected to VLAN2.

Another goal of the project was to capture alerts raised by the RUT. As the experiments were done on a specific RUT model, it was automated. The alerts are extracted from the RUT directly using a serial connection. This process is done in read-only way after finishing the simulation and affects neither traffic thoughput nor the RUT behaviour.

5.3 Architecture

5.3.1 Capturing

Traffic capturing is done using the *scapy* library directly wrapping its method for reading incoming packets from selected interface and writing them to a pcap file.

5.3.2 Preprocessing

Preprocess command cleans the pcap file from duplicated packets with the Linux *editcap* utility and transforms the pcap into an *output.json* file with all the sessions enlisted with their properties. Each session has a unique key identifying it and linking to its resources. Each session has a file with the contents from the traffic trace which are protocol specific and will be described in chapter 6. The auxiliary files are *output_grouped.json* and *output_sorted_sessions.json* to help the replay tool to load the sessions. The file names can be set to something different than “output”. A session is described primarily with the client and server IP address, session key and first packet time, and it also can contain other information about the client and server ports, flow size and so on.

5.3.3 Replay

The replay design is to have abstract clients and servers communicating via the RUT. Each client and server is represented by a separate running instance of some class able to implement the replayed protocol. The OD had two Ethernet interfaces: an internal one referred to as *eno1* in this work and external one called *enx*. The first was connected to the RUT LAN (VLAN1 of the intermediate switch) and the second one to RUT WAN (VLAN2).

One of the greatest troubles encountered during the project was replaying traffic from a single machine to both router interfaces. When a client and a server working with it are on the same machine, the traffic is sent through the Loopback interface and does not go out the machine. Setting up routes or *iptables* did not help. That is why it was decided to place clients and servers on two different virtual machines (VM). *Oracle VM VirtualBox* was used for that purpose. The IoT dvice is represented by one VM with Jessi OS - the simplest Linux OS found. It utilized the eno1 interface as its own. Server instances were created on another VM with Ubuntu OS, which uses the enx interface (fig. 4.4).

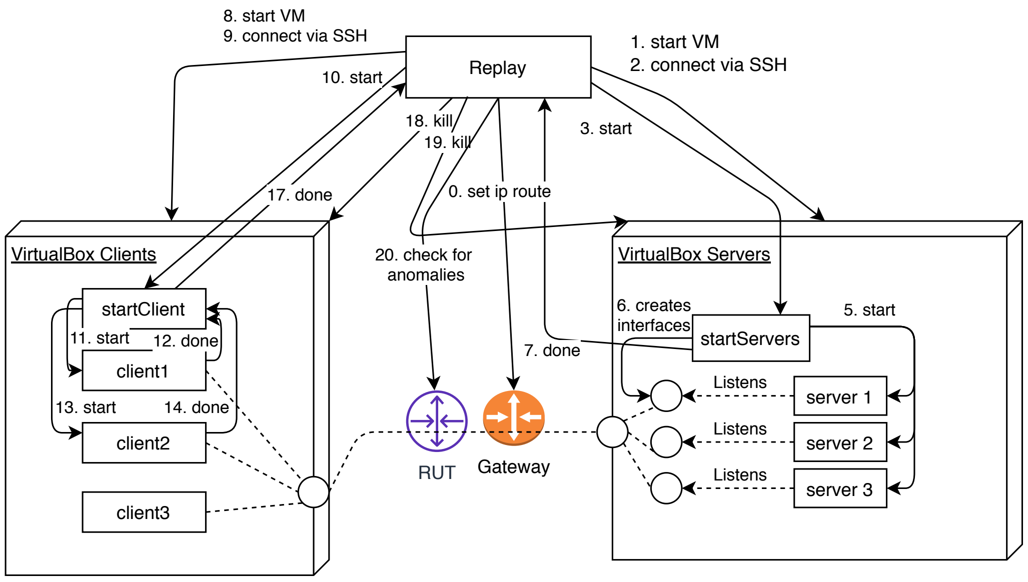


Figure 5.1: Replaying Traffic Flow

Vagrant is used to create the VM images during *make build*. It installs all the requirements on the VMs and sets up the connections for them. Vagrant is not required for each replay once the VMs have been created. *VirtualBox* provides a good command line interface for operating the existing images, which is utilized by the our testbed.

Once the replay tool started, first of all, it reads the json files input to it, enlists the servers, connects via SSH to the gateway router and routes the server IP addresses to the enx interface. This is done is done by *ip route* command. The whole process is automatic. The tool knows from the configuration that the enx interface needs to be used, finds out its IP address and uses it in the gateway.

Then the tool starts VMs, setting the MAC address of a client VM if required by the program input, uploads the chosen preprocessed files there by using a shared directory, connects to them via network interfaces, and runs the scripts start the servers and clients. The scripts running on the servers and clients are absolutely independent; the only link between those is the resource files created in preprocessing phase.

For each server host, a virtual interface on the VM is created having the actual IP address from the trace. This way, traffic routing is done automatically to those interfaces on arrival from the gateway to the server VM.

The servers are started all at once. Each protocol-host-port tuple is represented by one server instance. Each server is running as a separate process. Such a server loads all the information when started and listens to the corresponding interface (and port). When a connection comes to that listening socket, the server creates a new thread and answers the connection. The response depends on the server implementation. If the server has multiple sessions available for replay, it chooses the one corresponding to the requesting data and replies.

The clients program is designed in a different way. The script takes the session order from the preprocessed files and then replays each session one by one sequentially starting a new client instance for each session only after the previous client is terminated. Each client is running as a separate process also. In order to keep track on whether a client is done the termination events are put into a pipe.

After the clients program is finished, it releases an SSH terminal letting the root replay process on the OD know that the replay has finished. Then the replay tool turns off the VM and starts reading information from the RUT to report its reaction to the operator.

6

Protocol replay implementation

The components responsible for replaying certain network protocols are independent from the rest architecture and can be developed separately.

6.1 DHCP

Since the simulation happens on virtual machines, the OS handles the IP address assignment by itself. Thus, the developed testbed does not need to deal with the address configuration.

6.2 TCP

6.2.1 Preprocessing

The preprocessing phase utilizes the Linux tool *tcpflow* to extract traffic and *tstat* to get session parameters. The result of *tcpflow* is two files containing binary flows from the client to server and the other way around. These files are used as the source of data for TCP replay. The *tstat* tool extracts information about complete and incomplete TCP sessions into text files represented as tables. The testbed code parses those statistics files and translates the required information into output JSON files.

6.2.2 Replaying

The exact replaying of TCP is a difficult task with the original timing and fragmentation. Therefore, the TCP replay is done in such a way that the client sends the whole flow to a server and the server sends back the whole flow as response. After that, the communication is terminated.

Both the client and server utilise native OS sockets for sending information to relieve the program from tracking ports and IP addresses and the TCP logic. This means that the software is build on top of network transport layer.

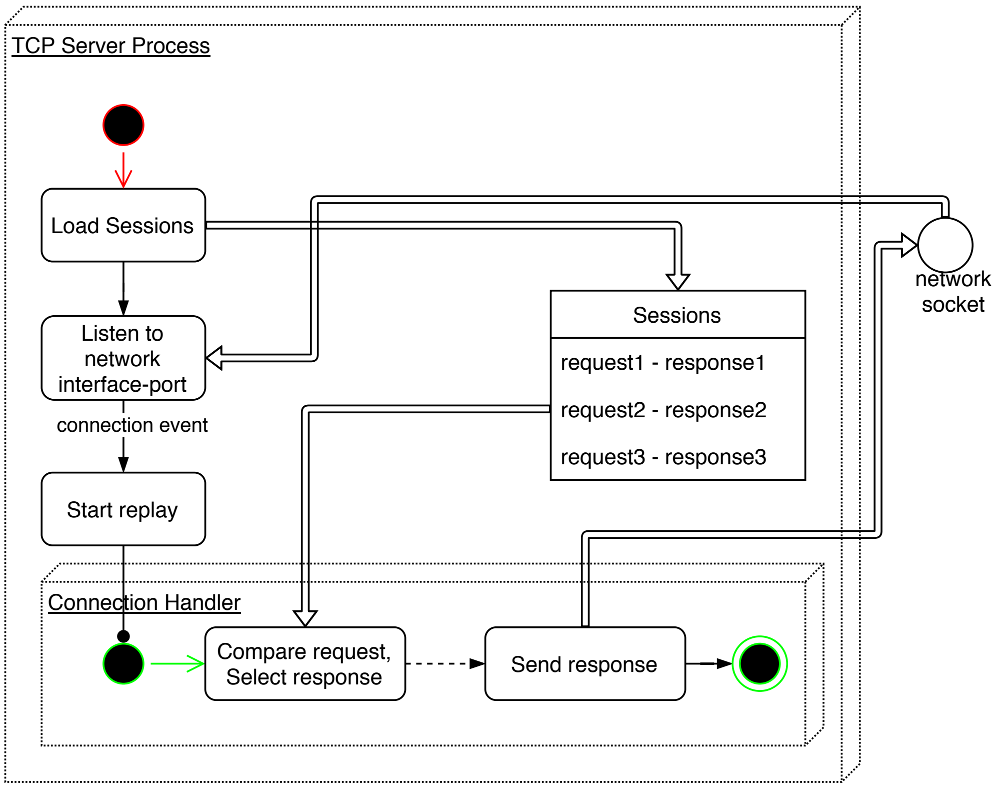


Figure 6.1: Replaying Traffic Flow

Often a server with the same IP addresses is engaged in several sessions in the original trace. All of these sessions are loaded to the replaying server. If the TCP server has multiple sessions, it compares the incoming flow against them and picks the session with the same client flow (fig. 6.1). If such a session is not found, the server picks the session corresponding to the current counter, which increases with each incoming connection. If the counter exceeds number of sessions, it is set back to zero.

6.3 UDP

6.3.1 Preprocessing

Just as TCP preprocessing, UDP preprocessing utilizes *tstat* and processes the traces in a similar way. The sessions are extracted within the developed tool, by selecting UDP packets from the preprocessed pcap file by the participating hosts and ports with the help of python *scapy* library.

6.3.2 Replaying

The client (fig. 6.2) utilises native OS sockets for sending the packets, working on top of transport layer. The program listens for packets from a specific source host and port. It uses *scapy* library to extract data from the pcap file and packets it contains. The client has a queue of incoming and outgoing packets for each session. If the current packet is supposed to outgoing for this client, the tool sends it to the defined receiver. If the packet is incoming, the tool listens to the connection within a certain timeout. If the expected packet does not arrive, the next packet is taken from the queue. When the queue is empty, the client terminates.

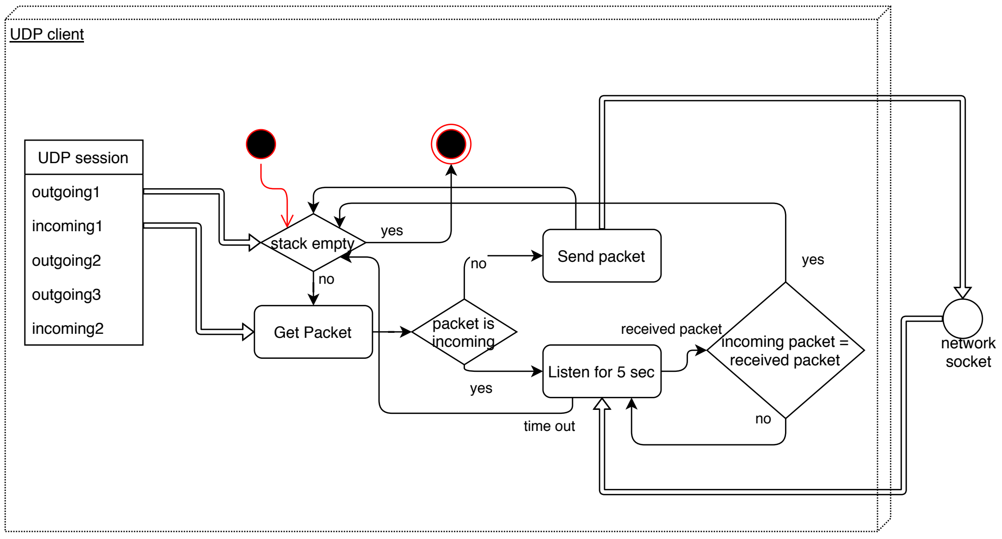


Figure 6.2: Replaying Traffic Flow

The server (fig. 6.3) is implemented in a similar way to TCP. It has a bunch of sessions related to it and listens to the socket corresponding to the correct interface and port. Once the socket gets a packet, the server selects a session for it. Then it creates a client (exactly the same as described above) in a separate thread and gives the selected session as queue.

6.4 ICMP echo

6.4.1 Preprocessing

Preprocessing is done using the *scapy* library. A session is identified by the echo destination IP address and the ICMP id field. The sessions are saved as pcap files containing the filtered packets.

6.4.2 Replaying

First of all, automatic echo is blocked on the OS level to ensure that the testbed controls which packets are replied.

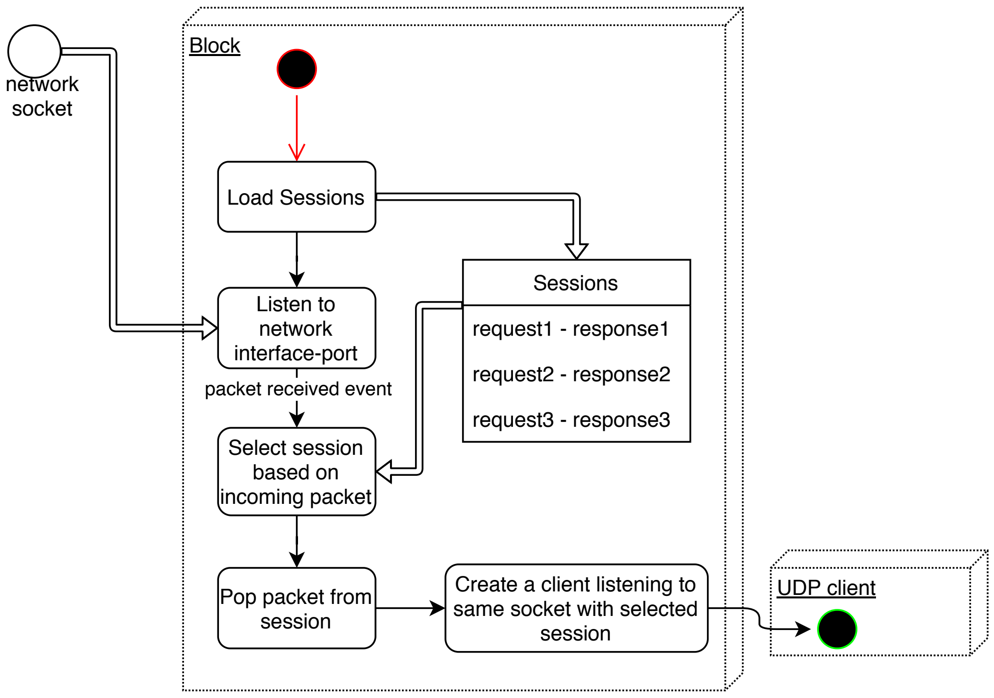


Figure 6.3: Replaying Traffic Flow

The server is determined only by the host IP address.

The client is implemented in a similar manner as UDP. The server, however, is very different (fig. 6.4). It does not create a separate thread for an incoming packet. Instead, it just picks a response by ICMP sequence number from a stored session. If the ICMP server does not find a corresponding request and response in stored the sessions, it echoes in the regular way. Both the client and server are utilising the *scapy* library to send the packets.

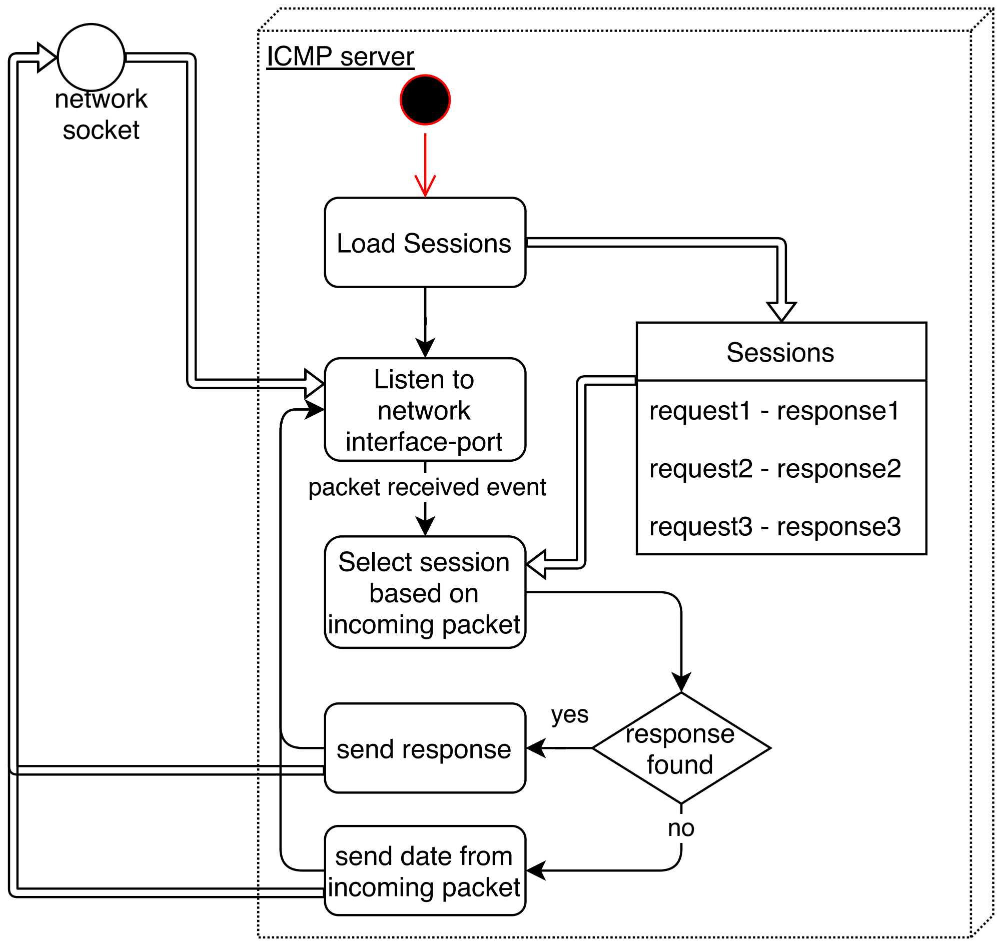


Figure 6.4: Replaying Traffic Flow

7

Discussion and framework analysis

This chapter reviews the system for possible alternative solutions, ways of improvement and comparison with other tools. Test results are also introduced here.

7.1 Device identification

The simplest way to identify a particular device is by its MAC address. So, a security router can learn what an IoT device under certain MAC address does in the network for some period. If this device starts operating in a way it never did before, the security router takes actions regarding the device.

At this point a doubt may appear if it is a good way to identify devices by their MAC addresses. Generally, the MAC address on an interface can be changed, unless the device OS or hardware disallows sockets that work at the link layer to overwrite the MAC address of the interface. This means that malware installed on an IoT device can use a different MAC address when networking. The counter argument measure from the security router could be forbidding new devices (identified by the MAC address) from connecting to the network without a user permission. However, it is common for many devices to use the ARP protocol. For example, this could be used to scan the network and to mimic another device using its MAC address. Preferably, the address would be spoofed for a PC that has complex behaviour from which it is difficult to detect anomalies. But the detection of anomalous device behaviour is out of the scope of this work.

7.2 Testbench setup

The capturing setup does not actually require a controlled gateway between the RUT and the outer network, along with the serial connection to RUT. The replaying setup does not require a mirroring switch because the OD LAN connection can go directly to the RUT. The server instances interface can go directly to the gateway router. Nevertheless, those devices were introduced in the architecture shown in figures 4.3 and 4.4 because installing and uninstalling devices would be unnecessarily time consuming for the operator.

With those devices untouched, the difference between the testbench states is the wired connection going from the OD *enx0* interface to VLAN2 instead of the mirroring port, not to mention the actual IoT device connected.

7.3 Experiments and results

The experiments for each step were executed using real IoT devices and a real RUT.

7.3.1 Capture

Traffic capturing was dispatched many times during the work on the project. It included traces from a personal computer, Arlo camera and Philips Hue smart light hub. Due to the ability of the tool to filter traffic, clean traces for IoT devices were saved to the OD hard drive as pcap files. The only anomaly caused by the mirroring switch was duplicating every packet in the files. However, this did not cause any trouble and can be eliminated with tools like *editcap*.

The traffic for the Arlo camera IoT device is described by the sample given in table 3.2. Overall, the camera uses HTTP and HTTPS to exchange data with its services, NTP for clock synchronization, and DNS for IP address discovery. This device was identified by the RUT during the traffic capture phase.

The Philips Hue hub also uses NTP, DNS, along with HTTP for communicating with Philips servers. In contrast to the Arlo camera, it uses the SSDP protocol (over UDP), and the most important difference was the smart lamp communicating with another local device (probably a mobile phone changing the light parameters) via TCP. This device was not identified by the RUT during the capture step.

A set of certain curl calls also triggered the RUT to identify the traffic source device as a fake smart lock for testing purposes. These traces including DNS and HTTP calls were also captured and saved.

7.3.2 Replay

The replay tool was used for all of the three cases described above. To keep the tests clean, each experiment happened with new MAC addresses given to the IoT simulating device.

The fake smart lock and the Arlo camera replayed traces triggered device identification in the RUT with the right device types for the corresponding MAC addresses. The smart lamp was not identified, however.

	Tcpreplay	web proxies	Marine	TCPopera	Our tool
Works through opaque proxy server	no	yes	no	no	no
Stateful / Network anomaly resistent	no	yes	yes	yes	yes
Packet time can be considered	yes	no	yes	approximately	no
Resources optimisation	-	yes	yes	yes	no
Protocols replayed	all	TCP	TCP	TCP	TCP, UDP, some ICMP
DNS dependency handling	no	yes	no	no	no

Table 7.1: Comparison with the similar replay tools

7.3.3 Anomaly

After the items were identified as IoT devices, unrelated traces were replayed on their MAC addresses. This triggered the RUT to alert about anomalies happening for those MAC addresses.

7.4 Comparison with other tools

The tool developed in this project is capable of testing routers with NAT. The tool supports multiple protocols and is extendable to more protocols. The summary of the capabilities, is shown in the comparison table 7.1

However, due to an extendable architecture with independent components, features like packet time keeping can be added, optimisations can be improved and the protocol list can be extended.

7.5 Future work

During this project a prototype that proved working was built. However, having more experience and given more time, there is has a vast space to extend and improve.

7.5.1 Wireless challenges

The mirroring switch alone is useless to capture traffic from a device that is connected to a router via Wi-Fi. In order to do that, either an intermediate access point or a wireless sniffer device would be required to capture radio signals. The latter would have problems if the radio is encrypted.

The same problem may potentially occur with replaying if a router expects a certain device type to be connected via Wi-Fi. This could also be solved by a wireless access point.

7.5.2 Resource consumption

As mentioned in chapter 6, the servers are loaded all at once. It allows the server VM to reply fast to any response, but makes it consume an unnecessary high amount of resources. Two improvements could be done. The easy one is to load sessions for a server only when it is required and to release the resource once the session is over. The more difficult one is to run the servers only when they are required. But the second solution would require very precise control of the ongoing processes, inter-OS pipes and non-trivial developer skills to ensure that connections are not dropped because the server is not ready.

An interesting challenge can arise for concurrent session replay. If you want to load the replaying instances in advance, the sessions can be blocked by dependency rules (described in section 3.4), and we do not know which next session is going to be unlocked affected by the replay process due to an ongoing session termination or another event introduced by a potential testbed.

7.5.3 Communication architecture

The client-server design for a session does not always reflect the reality. It might have problems with broadcast or other protocols. For example, an ICMP network error message may go beyond this design. It is sent not by the main session server but from some router. Then, we have a client receiving packets from two hosts.

The developed tool also does not support communication of multiple simulated devices in the RUT LAN, which is needed for the Philips Hue hub, for example.

7.5.4 Experiment analyzing

As mentioned in the experiment section, some of the devices going through the testbench may be unidentified. If the device type is identified while or after connecting it physically, but not identified when replaying it with the tool, a question arises why this happens. In order to be able to answer such questions more confidently, comparing the original and replayed traces would help. However, the tool developed in this project does not support saving the replay traffic, making it more difficult to analyze the replay step work and the results in general.

7.5.5 Inter-protocol dependencies

For this project, a TCP socket behaviour was not tested in the presence of ICMP error messages. According to the web discussion in [24], when receiving a fatal ICMP message, an OS TCP socket terminates automatically. However this may heavily depend on its implementation. Besides, UDP that is not stateful may react differently and requires a separate consideration.

The IP address dependency on DNS was not implemented. Potentially important situation here is DNS caching by the RUT. Not only it can lead to intercepting a request to the DNS server, but also result into an IP address different from the replayed trace! The preferable implementation would be to parse the DNS response on the fly and set the corresponding IP address in the whole process accordingly. This would also require reconsidering loading all the system on the start and make it more dynamic.

7.6 Testbench design alternatives

During this project multiple alternative solutions on some aspects were considered. Some of them are not feasible, others were to difficult to implement.

7.6.1 Ways of capturing traffic

One way to capture all the traffic going through a controllable router in the network is to open a shell on the RUT to execute tcpdump or a similar utility, running an extra process there, relying on the device memory and capacity, which is not a very clean way of testing it.

A cleaner way is to set transparent proxy devices around the RUT that captures the traffic. This does not affect the way RUT works but creates an overhead of complexity to control those devices.

The mirroring switch described in 5.1.2 is the simplest and the most effective way of capturing traffic.

7.6.2 Wired connections

From the operator perspective, having on the desk up to eight wires going in every direction, making it impossible to bunch them, along with power cables and the keyboard and key mouse wires was a little bit annoying. It became difficult to maintain them in order and especially to assemble the testbench.

As mentioned above, the switch is required only for capturing traffic. Removing it for replaying a device traffic, would reduce number of wires by three.

One of the ideas for reducing number of wires was to use a trunk connection going from the OD to the switch interface for accessing both VLANs. Packets going through the trunk have an additional VLAN layer between the transport and link layers stating which network to go through. However, this idea was rejected due to routing difficulties when replaying through one network interface, and bringing more complexity to maintaining this layer structure for OS network sockets.

7.6.3 Replaying instances

For two network interfaces with arbitrary IP addresses, belonging to different subnets built around different routers but located in the same OS environment, when sending a packet from one to another, it did not go through those routers but directly through the local OS subnet instead. This fact the traffic from going through the RUT. The attempted solutions to overcome the problem included using Ubuntu *route*, *iptables* and bridge tools. But none of the configuration attempts were successful. Although potentially those utilities could have solved the problem, the network stack in a single OD host has not been designed to support robust forwarding of packets through suboptimal routes. Using virtual machines worked better becaus each VM has its own network stack, and it is the only solution for this problem presented in this work.

Using virtual machines to keep the clients and the servers during replay phase was a working but costly solution causing a significant overhead in terms of the computer resources used, installation and loading time, and developed software complexity. It also potentially pollutes the replay, since the installed OS can have traffic of its own.

8 Conclusion

Many IoT devices are vulnerable and require additional protection that can be granted by a firewall, IPS or IDS.

A framework replaying traffic is important for end-to-end testing of network security devices varying from simple firewalls to complex IDS. However, the same trace can be replayed in different manner, thus it is important to know what properties of the traffic are required to be maintained in replay process and the tool should be adjusted.

For a consistent replay different traffic dependencies need to be considered, since data like a port or an address defined in one place can affect another place which is not connected session or protocol-wise. And this data can differ in the original and the replayed traffic.

Traffic replay can happen at a different network layer. The choice of the layer should be done based on the replay purpose — what traffic characteristics described in section 3.1 are wanted to be preserved. The lower layer is picked, the more characteristics tool can maintain, but the more complicated software needs to be developed, which would actually duplicate the higher layers' functionality.

The testbed should have separate steps of capturing and replaying traces. Putting any intermediate trace modifying or preprocessing into a separate step simplifies software and saves time during replay step.

The software developed in this project proved to be working on a real RUT and real traces. It is capable of capturing, modifying and replaying traffic. As a result of experiments, the virtual client instances simulating the real devices have been identified as the fake test smart lock and real Arlo camera. After adding unrelated traces into the existing traces, the RUT indicated anomalies in the traffic produced during the simulation of the identified devices.

Since one of the project targets was replaying multiple network protocols, it is feasible to split the tool into independent components. The developed tool replays TCP, UDP and ICMP echos. But the list can be extended to other higher or lower layer protocols like Ethernet to cover more potential traffic or http to maintain application layer consistency.

References

[1] Mar. 2020. [Online]. Available: https://www.tcpdump.org/manpages/tcpdump.1.html (visited on 26/07/2020).

[2] K. Angrishi, 'Turning internet of things(iot) into internet of vulnerabilities (iov) : Iot botnets', 2017. [Online]. Available: https://arxiv.org/abs/1702.03681.

[3] M. Abu Rajab, J. Zarfoss, F. Monrose and A. Terzis, 'A multifaceted approach to understanding the botnet phenomenon', in *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '06, Rio de Janeriro, Brazil: Association for Computing Machinery, 2006, pp. 41–52, ISBN: 1595935614. DOI: 10.1145/1177080.1177086. [Online]. Available: https://doi.org/10.1145/1177080.1177086.

[4] [Online]. Available: https://cleantalk.org/blacklists.

[5] S. Manoharan and S. Rathi, 'Iot malware : An analysis of iot device hijacking', Sep. 2019. [Online]. Available: https://www.researchgate.net/publication/335676455_IOT_Malware_An_Analysis_of_IOT_Device_Hijacking.

[6] S. Shirali-Shahreza and Y. Ganjali, 'Protecting home user devices with an sdn-based firewall', *IEEE Transactions on Consumer Electronics*, vol. 64, no. 1, pp. 92–100, 2018.

[7] P. Avishai Wool, *Packet filtering and stateful firewalls*, Jan. 2012. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.146.5044&rep=rep1&type=pdf (visited on 26/07/2020).

[8] B. Soewito and C. E. Andhika, 'Next generation firewall for improving security in company and iot network', in *2019 International Seminar on Intelligent Technology and Its Applications (ISITIA)*, 2019, pp. 205–209.

[9] R. T. El-Maghraby, N. M. Abd Elazim and A. M. Bahaa-Eldin, 'A survey on deep packet inspection', in *2017 12th International Conference on Computer Engineering and Systems (ICCES)*, 2017, pp. 188–197.

[10] C.-Y. Huang, Y.-D. Lin, P.-Y. Liao and Y.-C. Lai, 'Stateful traffic replay for web application proxies', en, *Security and Communication Networks*, vol. 8, no. 6, pp. 970–981, Apr. 2015, ISSN: 1939-0122. DOI: 10.1002/sec.1053. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1053 (visited on 06/11/2018).

[11] S. Somani, P. Solunke, S. Oke, P. Medhi and P. P. Laturkar, 'Iot based smart security and home automation', in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, 2018, pp. 1–4.

[12] F. E.-M. 1. D. Bastos M. Shackleton 1, 'Internet of things: A survey of technologies and security risks in smart home and city environments', 2018. [Online]. Available: https://digital-library.theiet.org/content/conferences/10.1049/cp.2018.0030.

[13] 2020. [Online]. Available: https://iperf.fr/iperf-doc.php (visited on 26/07/2020).

[14] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. USA: Cambridge University Press, 2008, ISBN: 0521880386.

[15] 2020. [Online]. Available: https://tcpreplay.appneta.com/wiki/overview.html.

[16] G. G. X. Thomas S. Le Vier, 'A tool for stateful replay', en, 2013. [Online]. Available: http://www.dtic.mil/dtic/tr/fulltext/u2/a579972.pdf (visited on 06/11/2018).

[17] S.-S. Hong and S. F. Wu, 'On interactive internet traffic replay', in *Recent Advances in Intrusion Detection*, A. Valdes and D. Zamboni, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 247–264, ISBN: 978-3-540-31779-1.

[18] M. Miettinen, S. Marchal, I. Hafeez, N. Asokan, A. Sadeghi and S. Tarkoma, 'Iot sentinel: Automated device-type identification for security enforcement in iot', *CoRR*, vol. abs/1611.04880, 2016. arXiv: 1611.04880. [Online]. Available: http://arxiv.org/abs/1611.04880.

[19] 1981. [Online]. Available: https://tools.ietf.org/html/rfc793.

[20] 2020. [Online]. Available: https://tcpreplay.appneta.com/wiki/tcprewrite-man.html.

[21] A. S. Tanenbaum and D. J. Wetherall, *Computer Networks*, 5th. USA: Prentice Hall Press, 2010, ISBN: 0132126958.

[22] Feb. 2009. [Online]. Available: https://tools.ietf.org/html/rfc5461.

[23] Sep. 1981. [Online]. Available: https://tools.ietf.org/html/rfc792.

[24] Jun. 2004. [Online]. Available: https://www.dslreports.com/forum/r10430146-What-ICMP-error-does-TCP-socket (visited on 26/07/2020).