universidad de león

Departamento de Matemáticas

Máster Universitario en Investigación en Ciberseguridad

Trabajo Fin de Máster

Exfiltración de datos en protocolos IoT

Data Exfiltration in IoT Protocols

Autor: Daniel Uroz Hinarejos
Tutor: Ricardo J. Rodríguez

(Septiembre, 2020)

Universidad de León
Departamento de Matemáticas
Máster Universitario en Investigación en Ciberseguridad
Trabajo Fin de Máster

Alumno: Daniel Uroz Hinarejos
Tutor: Ricardo J. Rodríguez

Título: Exfiltración de datos en protocolos IoT

Title: Data Exfiltration in IoT Protocols

Convocatoria: Septiembre, 2020

Resumen:
Se denomina exfiltración de datos a la transferencia no autorizada de información entre sistemas. Debido a que gran parte de las empresas despliegan algún tipo de mecanismo de defensa (como cortafuegos), los adversarios intentan ocultar que la transferencia de información está teniendo lugar gracias al uso de canales encubiertos. Un canal encubierto es cualquier comunicación para transferir información de forma que viole las políticas de seguridad de los sistemas. El ejemplo más común de canal encubierto es el de encapsular un protocolo de red sobre otro, siendo el último de ellos el canal encubierto.

Las redes de Internet of Things (IoT) despliegan varios sensores, actuadores, objetos y nodos inteligentes capaces de comunicarse entre ellos sin intervención humana. Comúnmente, estos dispositivos generan información que posteriormente se envía a otra red encargada de recolectar y procesar toda esta información. Para que los dispositivos se puedan comunicar, utilizan protocolos IoT especialmente diseñados para dispositivos de poca capacidad computacional.

En este estudio, se ha investigado cómo se pueden utilizar los protocolos IoT para la exfiltración de datos. Concretamente, se han estudio los protocolos de capa de aplicación *Constrained Application Protocol* (CoAP), *Message Queuing Telemetry Transport* (MQTT), y *Advanced Message Queuing Protocol* (AMQP). Primero, se ha definido teóricamente cómo se pueden utilizar para la exfiltración de datos y, posteriormente, se ha comparado su rendimiento sobre los protocolos tradicionales. Finalmente, se presenta la librería `chiton` diseñada para la exfiltración de datos a través de protocolos IoT, y se ha medido empíricamente el tiempo de exfiltración para distintos tamaños de ficheros.

Palabras clave: Exfiltración de datos, Protocolos IoT, CoAP, MQTT, AMQP

Agradecimientos

Tras más de 5 meses de dedicación a este trabajo, me gustaría recapitular y escribir unas palabras para aquellas personas que han hecho posible que no desistiera, o no del todo.

En primer lugar, agradecer a Ricardo su asesoramiento y dedicación por centrarme y no permitir que me diluya en un mar de chorradas, como suele ser costumbre.

En segundo lugar, agradecer tanto a Alejandro, Mario, Rubén, y Saúl por conseguir que por muy difícil que sea la situación siempre consigamos banalizarla; como a Alfonso, Iñaki, José Cristian, Juan, y Ramón por haber estado siempre ahí, a pesar de no ser capaces de organizarnos y vernos más que una vez al año.

En último lugar, agradecer a mi familia su apoyo incondicional desde siempre, a pesar de mi carácter y de las adversidades.

Gracias.

P.D: Por arruinarme todos los planes y obligarme a sacar tiempo para este trabajo, sin tu ayuda tampoco hubiese sido posible, maldito viruse.

Data Exfiltration in IoT Protocols

ABSTRACT

Data exfiltration is the unauthorized transfer of information from one information system to another. As mostly enterprises deploy some sort of defense mechanism (like a firewall), adversaries try to mask that transference is taking place using covert channel techniques. A covert channel is any communication to transfer information in a manner that violates the systems security policy. The most common example is to *tunnel* one protocol into another, being the latter the covert channel.

Internet of Things (IoT) networks deploy various sensors, objects and smart nodes that are capable of communicating with each other without human intervention. Most commonly, these *things* generate information that is lately sent to an outside network responsible of recollection and processing. In order to communicate, they rely on IoT protocols specially design for constrained devices.

In this study, we investigated how IoT protocols can be used for data exfiltration. Namely, we focused on application layer protocols such as Constrained Application Protocol (CoAP), Message Queuing Telemetry Transport (MQTT), and Advanced Message Queuing Protocol (AMQP). First, we theoretically define how they can be used to exfiltrate data and, second, we compare how they perform over traditional protocols. Finally, we present the Python library `chiton`, designed to exfiltrate data trough IoT protocols, and we empirically measured the data exfiltration elapsed time for different amounts of data.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Information is usually described as one of the most valuable resources in the digital era. This means that criminals are also interested in gather information, either to sell it in the dark market or to gain any type of advantage against their competitors (in case of not so legitimate companies, or state-sponsored groups). As pointed by [1], external actors are responsible for an increasing percentage of data theft, rising from 57% of breaches in 2015 to 61% in 2018.

Hence, computers and networks within organizations are a common target for several types of attacks, varying from Denial of Service (DoS) to phishing. As attacks have become more sophisticated, defenses also needed to evolve. As such, firewalls, demilitarized zone (DMZ) networks, or Intrusion Detection Systems (IDS) are (or should be) techniques widely deployed in enterprise networks.

Unfortunately, in the case of all these countermeasures failed for whatever reason (e.g: misconfiguration), criminals are free to collect as much information as possible and, then, *exfiltrate* all collected data. The concept of exfiltration is inherited from military and, in the context of computer security, it refers to the unauthorized transfer of information from an information system [2].

In order to prevent such misbehavior, some solutions like Data Loss Prevention (DLP) systems aim to detect when data is leaving organization network. For example, DLP can be configured to search for credit card numbers in a company private network, and block them if they are spotted in the network traffic before being sent to criminals.

To overcome these limitations, adversaries use techniques such as *covert channels* to avoid detection. Covert channels, first described in [3], have been defined as any communication channel that can be exploited by a process to transfer information in a manner that violates the systems security policy [4]. That means, for example, that adversaries can use well-established covert channels in Domain Name System (DNS) to bypass protections, as DNS is allowed by firewalls in most enterprise networks [5].

Defending computer assets gets worse when these assets are resource constrained because little security systems can be deployed, as in the case of Internet of Things (IoT). Without a consensus of a proper definition, IoT is usually described as the integration of various sensors, objects and smart nodes that are capable of communicating with each other without human intervention [6].

More and more IoT devices are expected to be incorporated of every aspect

of our daily life, from voice assistants to critical infrastructure actuators, until reaching 21.5 billion devices by 2025 [7]. With the precedence of Mirai botnet, which has been shown one of the largest Distributed Denial of Service (DDoS) up to date [8], studying implications of IoT devices and networks is a key aspect for a more secure world.

Whereas traditional covert channels for data exfiltration are well-grounded [9, 10], there is little literature discussing their characteristics in IoT networks [11, 12, 13]. Moreover, to the best of our knowledge, there is no extensive comparative of different network protocols, either traditional or IoT specific. Therefore, the contribution of this work is four fold:

1. In addition to traditional data exfiltration techniques, we identify and review IoT protocols suitable to data exfiltration. Namely, we studied Constrained Application Protocol (CoAP), Message Queuing Telemetry Transport (MQTT), and Advanced Message Queuing Protocol (AMQP).
2. To fill the existing gap in literature, we extensible compare all selected protocols based in characteristics such as overhead and useful payload for network packet. Additionally, we compare the time to exfiltrate different amounts of data, in order to highlight those types of packets theoretically valid but unfeasible in use case real world scenarios.
3. We added support for the AMQP protocol to the open-source Scapy Project[1].
4. Finally, a proof-of-concept of data exfiltration using these IoT protocols is presented and evaluated.

1.1 Objective

This document aims to define and compare data exfiltration techniques within IoT networks. Additionally, a proof-of-concept of different types of data exfiltration techniques in IoT is presented. The research method includes the following steps:

- Study of data exfiltration methods and countermeasures.
- Identification of IoT network protocols suitable to data exfiltration.
- Extensive comparison of selected protocols.
- Analysis, design, implementation, and test of the exfiltration library.

The library `chiton` was developed as a side-product of this research process. This library is licensed under GNU GPL v3, and it can be accessed alongside its documentation in:

Chitón is the Spanish interjection for *hush*

`https://github.com/duroz/chiton`

[1] Scapy Project: `https://scapy.net/` (visited on 2020/08/28)

1.2 Structure

The rest of this document is structured as follows. Chapter 2 contains a general overview of data exfiltration techniques and presents IoT protocol characteristics. Chapter 3 presents the related work regarding covert channels techniques, and data exfiltration related work. Chapter 4 shows a comprehensive description of characteristics for data exfiltration, and a theoretical comparison within all studied protocols. Chapter 5 presents the `chiton` Python library. Experimentation and evaluation of the library are shown in Chapter 6. Finally, Chapter 7 exposes final thoughts and conclusions of this study.

Additionally, Appendix A shows a detailed view of invested hours in this work.

Chapter 2

Previous Knowledge

This chapter describes the context to a better understanding of the study carried out. First, a comprehensive view of data exfiltration techniques is presented. Then, a description of the most relevant aspects of IoT in the context of exfiltration is described.

2.1 Data Exfiltration

As pointed out previously, data exfiltration is the unauthorized transfer of information from an information system [2]. Data exfiltration *per se* does not necessarily need to be any advanced technique, as a simple outbound HTTP connection from a compromised email server would accomplish this task. Nevertheless, when some type of defense mechanism as firewalls is deployed within a network, an adversary usually needs some technique to overcome these defenses.

Steganography

One solution of the problem is what is called *steganography*, a term which comes from ancient the Greek words *steganós* (στεγανός), meaning covered or concealed, and *graphia* (γραφή), meaning writing; that is, to hide a subliminal message into another medium. As the etymology suggests, first known cases go back from ancient Greece, when secret messages were tattooed in the shaved head of messengers, and their hair was left to grow again to bypass any guard check and deliver the secret message.

In the context of computing, steganography (also called *data or information hiding*) applies to the art of concealing a text, image, video, etc. into another medium, either equal or different. One example of the most used techniques is called Least Significant Bit (LSB) steganography, which consists in hide information by modifying the least significant bit of encoding (e.g: image pixels) to make changes imperceptible to anyone but those knowing this technique is applied, in favor of a little medium degradation [14]. This way, steganography provides security through obscurity.

Within the context of data exfiltration, steganography is not a exfiltration technique itself, as it relies on any other communication channel to transmit

the hidden information.

Covert Channels

Covert channels are described as any communication channel that can be exploited by a process to transfer information in a manner that violates the systems security policy [4]. Covert channels were first described in [3], and they are a common solution to the **warden problem** [15], which states the following:

> *Two accomplices in a crime have been arrested and are about to be locked in widely separated cells. Their only means of communication after they are locked up will he by way of messages conveyed for them by trustees –who are known to be agents of the warden. The warden is willing to allow the prisoners to exchange messages in the hope that he can deceive at least one of them into accepting as a genuine communication from the other either a fraudulent message created by the warden himself or else a modification by him of a genuine message. However, since he has every reason to suspect that the prisoners want to coordinate an escape plan, the warden will only permit the exchanges to occur if the information contained in the messages is completely open to him –and presumably innocuous. The prisoners, on the other hand, are willing to accept these conditions, i.e., to accept some risk of deception in order to be able to communicate at all, since they need to coordinate their plans. To do this they will have to deceive the warden by finding a way of communicating secretly in the exchanges, i.e., of establishing a "subliminal channel" between them in full view of the warden, even though the messages themselves contain no secret (to the warden) information. Since they anticipate that the warden will try to deceive them by introducing fraudulent messages, they will only exchange messages if they are permitted to authenticate them.*

Covert channel are distinguish from stenography due to the former require some sort of communication as cover, and the latter require some sort of content as cover. Historically, there is a dual distinction when studying covert channels: (1) **storage covert channels**, which use values written/read in transmitted objects (e.g: unused header fields); and (2) **timing covert channels**, which use time of transmitted objects to module information (e.g: delayed packet means bit 1, and on-time packet means bit 0) [16].

Simultaneously, there is a distinction between three types of wardens [17]. A **passive warden** only spies on the communications channel between the prisoners. An **active warden** modifies the data being sent between the prisoners without changing semantics. Finally, a **malicious warden** alters the prisoners' messages without impunity, but this type of warden is not usually used in reality [10].

Note that whereas in the traditional definition of covert channel, receiver and sender must agree in the used technique; within the context of data exfiltration, the sender and receiver are usually the same individual.

Tunneling

Tunneling can be seen as a specific type of storage covert channel, where one protocol is embedded inside the payload of another protocol. This tech-

nique allows to bypass firewall protection where one protocol is allowed for outcome connections and another one is blocked; in this case, the former is the *tunnel* protocol and the latter is the *payload*. One of first examples of this technique is Project Loki, which tunnels communication into the payload of ICMP echo messages [18], a kind of traffic which is usually allowed into most network as it is seen as innocuous. Although tunneling relies on storage covert channel, it tries to maximize throughput instead of maintaining a low profile communication (that is, to remain undetected as long as possible) [10].

Side Channels

Whereas side channels are closely related, strictly speaking they are not a data exfiltration technique. Side channels transfer information in an *unintended* way. For example, electromagnetism can leak the secret key generated from a cryptographic key generation algorithm. Hence, this type of techniques are out of scope of this study.

2.2 Internet of Things (IoT)

IoT is the integration of various sensors, objects and smart nodes that are capable of communicating with each other without human intervention [6]. IoT devices can be usually be divided between actuators and sensors. IoT sensors gather information that is later transmitted to a control center, which takes decisions based in this information and can trigger some sort of physical intervention carried out by an IoT actuator.

IoT devices greatly vary in capabilities and technologies. In order to communicate each other, these devices use protocols specifically designed to meet the requirements of constrained devices. As a process of standardization, the IoT protocol stack [19] is divided from down to upper into six layers:

- *Physical*: this layer defines how the bits need to be transmitted over a physical data link connecting network nodes. For example, IEEE 802.15.4 is a physical layer standard specially designed for low-rate wireless personal area networks.
- *Medium access control* (MAC): this layer defines how the hardware interacts with the transmission medium. For example, IEEE 802.15.4e is the part of the IEEE 802.15.4 standard responsible to describe this interaction.
- *Adaptation*: this layer is responsible of disassembling and assembling packets into fragments before and after transmission. For example, 6LoWPAN defines a IPv6 over low power wireless personal area networks.
- *Network*: this layer is responsible for routing the packets. For example, ROLL RPL is a network protocol for Low Power and Lossy Networks.
- *Transport*: this layer provides communication between devices for the upper layer. In IoT constrained devices, UDP is preferred over TCP due to its little overhead thanks to connectionless communication.
- *Application*: this layer is responsible for data transmission and representation. Some examples are Constrained Application Protocol (CoAP), Mes-

sage Queuing Telemetry Transport (MQTT), Advanced Message Queuing Protocol (AMQP), Extensible Messaging and Presence Protocol (XMPP), to name a few.

IoT devices are usually deployed in isolated networks, even geographically distant thanks to low-power wide-area networks, where devices can be accessed from kilometers away. An IoT network needs an IoT gateway to be able to communicate over Internet and, in some configurations, to an IoT cloud provider. Some examples of these IoT protocols to connect to the cloud are MQTT 3.1.1 for AWS IoT Core, Google Cloud IoT Core, and Microsoft Azure IoT Hub; the latter also supports AMQP 1.0.

Chapter 3

Related Work

To the best of our knowledge, previous works consist in a selection and description of covert channels, but not within the context of data exfiltration. Besides, this is the first study analyzing and comparing different IoT protocols for data exfiltration. Additionally, we propose the use of non-previously analyzed protocols such as MQTT and AMQP.

3.1 Covert Channels

Covert channels have been widely studied since firstly described in [3]. First of all, covert channels were defined just for transmitting information between processes resident in the same computer [3]. With the rise of interconnected computers, cover channels were ported to network protocols, first to LAN networks [16] and, finally, to the Internet [9].

Regarding protocols, *optional fields* of the TCP/IP protocol suite like IP packet identification, TCP initial sequence number, and TCP acknowledged sequence number were used to transmit information in [9]. With the upgrade of Internet Protocol (IP) to version 6, the use of other fields such as IPv6 extension header has arisen too [20]. A more in-deep study can be found in [21], where 22 storage covert channels in IPv6 were presented and analyzed in the context of active wardens (stateless, stateful, and network aware).

Through the years, several cover channels and steganography studies contemplate their use in a variety of protocols such as HTTP, HTTPS, WLAN, VoIP, SSH, FTP, NTP, to name a few. These studies have been collected in complete surveys covering different network protocols, techniques, and countermeasures [10, 22, 23]. More advanced cover channel have been also proposed, like manipulation of the physical layer of IEEE 802.3 10 Gigabit Ethernet [24] or covert channels between virtualized systems in the cloud [25].

Tunneling protocols into Internet Control Message Protocol (ICMP) was firstly presented in [18]. Consequently, Domain Name System (DNS) were also used to tunnel protocols [26], being iodine the most matured and studied tool [27], which uses TXT, SRV and NULL resource records for tunneling. In a more general vision, there are studies discussing the use of cover channels in malware communications [28, 29].

Regarding IoT, an analysis of covert channels for CoAP is presented in [13].

A study of covert channels in Wireless Sensor Networks can be found in [30], where the modulation of transmission power and modulation of sensor data are used to covert transmission. The use of timing covert channel thanks to different paths taken by packets to reach destination in the network topology is proposed in [11]. We can also found a classification of attack patterns for IoT environments in [31]. Finally, a survey of steganography, covert channels and future work in IoT can be also found in [12].

3.2 Data Exfiltration

Most of the data exfiltration studies are focused in air-gapped covert channels, that is, covert channels specially tailored to overcome the gap of physical-isolated system. In this regard, an acoustic-based data exfiltration based in hard-drive noise is presented in [32]. A malware proof-of-concept for data exfiltration using visual channels via a Rapsberry Pi is presented in [33]. The same visual concept is used in [34], but using Bluetooth IoT bulbs instead. Finally, there is a comparative of data exfiltration rates using radio frequencies is also introduced in [35].

Regarding detection, a new model to detect data exfiltration over the DNS protocol is presented [36], which proposes a solution to detect low-thought DNS data exfiltration. Likewise, the work in [37] states that their deep learning-based solution has a relatively low precision for data exfiltration and keylogging attacks. A similar hypothesis is supported in [38], which presents a Bot-IoT dataset concluding that data exfiltration has the worst metric of all types of attacks contained in the dataset.

Publicly available tools to perform data exfiltration are intended to test implemented Network Monitoring and Data Leakage Prevention configurations. The two most relevant examples are: (1) `DET`[1], which supports HTTP, HTTPS, ICMP, DNS, SMTP/IMAP, FTP, and SIP protocols; and (2) `PyExfil`[2] which supports DNS, ICMP, NTP, BGP, QUIC, Slack, POP3, FTP, IP, and HTTP/S protocols, as well as some physical (e.g: audio) and steganographycal techniques. Both projects are standalone tools, which means that you are tied to the way they operate. However, `chiton` is a modular library designed to be imported in any project. The library is only responsible to encapsulate any binary data into protocol packets and, if necessary, send them over the network.

[1] DET: https://github.com/PaulSec/DET (visited on 2020/08/09)
[2] PyExfil: https://github.com/ytisf/PyExfil (visited on 2020/08/09)

Chapter 4

Exfiltration Analysis

In this chapter, we model the adversary characteristics and placement within a typical IoT corporate network layout. Additionally, we analyze the interesting protocol characteristics for data exfiltration, and we describe all selected protocols from this point of view. Finally, we present a comprehensive comparative between protocols.

4.1 Adversary Model

The corporate network presented in this case is divided in two subnetworks: an *IoT heterogeneous network*, in which IoT devices are mixed with operator workstations connected to an IoT gateway; and the IoT cloud network, responsible of long-term storage and data analysis. The IoT gateway allows to connect IoT devices through Internet to instances in the cloud service provider network, as shown in Figure 4.1.

We distinguish between different level of adversaries' expertise, from basic to more advanced:

- **Basic adversary**: there is one compromised system inside the corporate network. An example scenario is an adversary who gains access to a workstation thanks to stolen credentials (Figure 4.2).
- **Advanced adversary**: in this scenario, the adversary compromises the IoT gateway responsible to route all IoT protocols over IP to the Internet. Therefore, the adversary can sniff all incoming traffic and duplicate it to a server of his own (Figure 4.3).
- **Advance Persistent Threat (APT) adversary**: the adversary has instances in the same cloud network provider, and the path from the IoT gateway to the legitimate provider is also compromised to change Border Gateway Protocol (BGP) protocol to forge routing to another instance within the cloud (Figure 4.4).

Complementing this vision, we propose a distinction between **stealthy and rough adversary**. A stealthy adversary tries to adequate outgoing packets to commonly used sizes, whereas a rough adversary maximize the possible payload for every packet. We refer to them simply by adversary when one technique can be used for both types. Regarding this distinction, the payload of

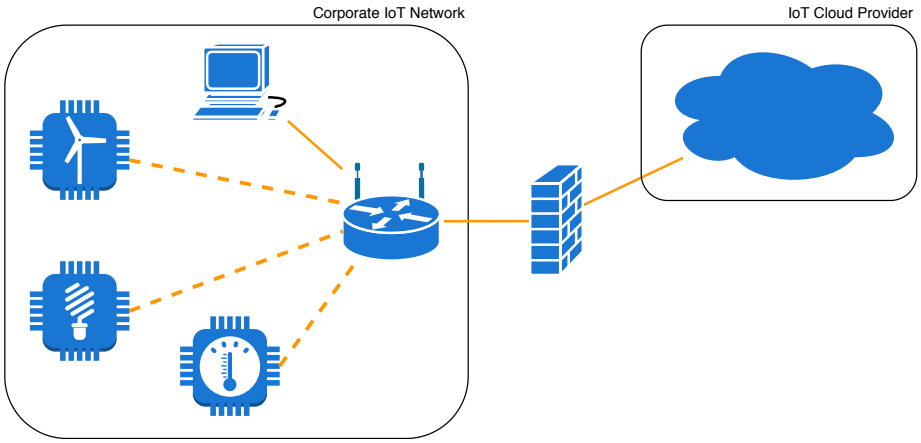


Figure 4.1: Example of IoT network distribution.

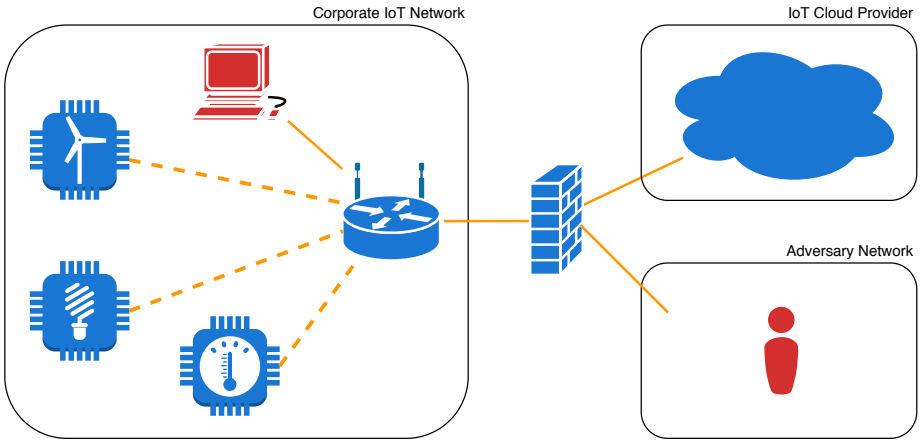


Figure 4.2: Basic adversary model.

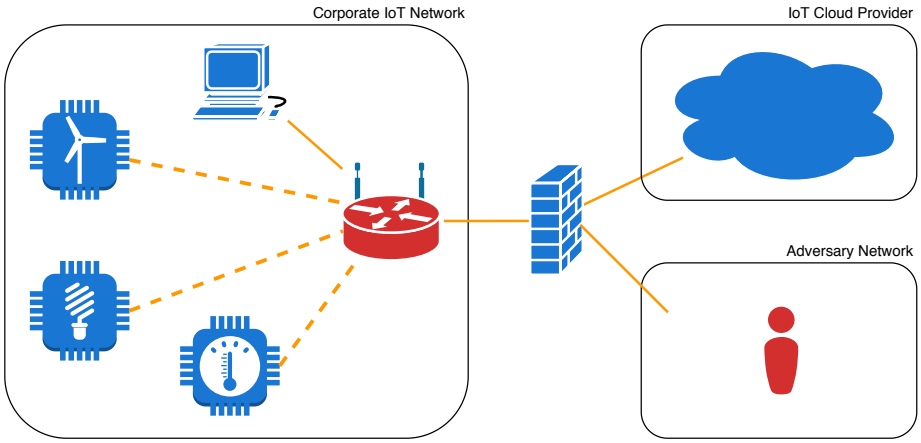


Figure 4.3: Advanced adversary model.

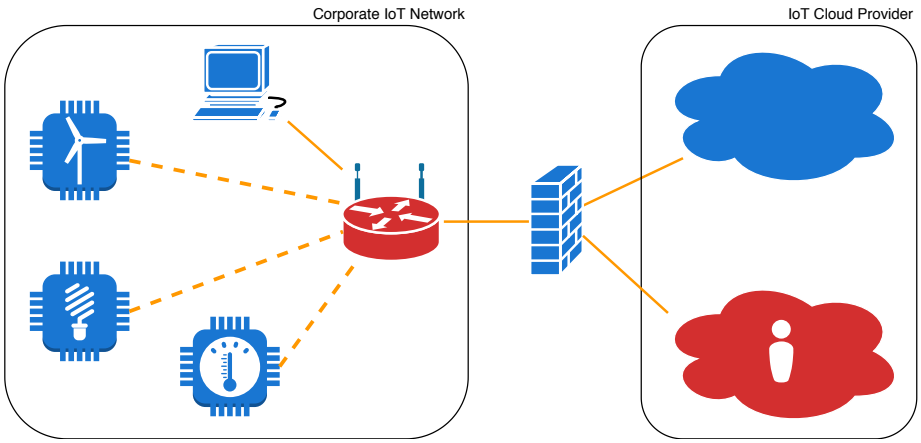


Figure 4.4: APT adversary model.

ICMP usually has a length of 56 bytes, but its theoretical payload can be up to 65,493 bytes (`0xffff` IP packet limit minus headers), so a stealthy adversary adequates its traffic to the lowest (64 bytes) to impersonate ping traffic. In general, more deviated the network packet size is from the typical size, the easier to spot such a network packet as malicious.

In this scenario, the attack has no privileged access in any compromised device, so no port below 1024 can be opened to receive or send data. This constraint limits the type of data exfiltration techniques available, because the adversary usually needs to act as a client, and thus limiting the type of protocol packets available to use as tunnel. For example, whereas a client can only send a DNS request with a payload up to 245 bytes, the server's response can contain up to the theoretical limit of 65,156 bytes. This is the reason behind the limitation of DNS tunneling tools such as `iodine`, which states in its official webpage[1] that "*the bandwidth is asymmetrical with limited upstream and up to 1 Mbit/s downstream*".

4.2 Characteristics of Protocol Exfiltration

In the context of data exfiltration, a protocol can be described based in three main characteristics:

1. **Packet type**: each protocol defines a series of packet types, each one tailored for a specific purpose. Depending of the type, it can be classified as:
   (a) **Payload**: how much data a protocol is able to carry in a single packet. Some packets can have none, like a opening connection packet type, or it can carry a lot of data thanks to a transfer type packet.
   (b) **Overhead**: every byte not representing the actual data to exfiltrate. Some packets can have larger headers unsuitable to embed data

[1] iodine: `https://code.kryo.se/iodine/` (visited on 2020/08/29)

onto it, while others can have just few bytes indicating than the rest of the payload is binary data application-specific.

2. **Transport**: depending on the protocol, it can rely on a connectionless transport layer like UDP, which is lightweight for constrained devices but it incurs in a probability of lost. On the contrary, a connection-oriented protocol like TCP can add more latency thanks to error detection or retransmission for packet reliability.

3. **Error detection**: as exfiltration consists in sending data over a network, a desirable characteristic is to count on any type of checksum redundancy to spot when received data have been corrupted.

Additionally, we need to distinguish between theoretical and practical payload size. Whereas a specific standard may not impose any type of limitation for certain field size, due to some implementations constraints, some sizes could be much lower that those specified by the standard in a real world scenario.

4.3 Traditional Protocols

In this section, we discuss traditional protocols suitable to data exfiltration. We call traditional protocols to those included in the Internet protocol suite and usually allowed in all networks, regardless of its nature. Namely, we focus on Internet Control Message Protocol (ICMP), Network Time Protocol (NTP), and Domain Name System (DNS).

In the sequel, we describe the packets suitable to data exfiltration for each protocol, their characteristics, their supported data types, and their limitations (if any).

4.3.1 Internet Control Message Protocol (ICMP)

ICMP is a network-layer (layer 3 in OSI model) protocol designed to report network communication issues, for example, for when a datagram cannot reach its destination [39]. The same protocol is also used to perform network diagnostics thanks to *Echo Reply/Echo Request* packets, both used by the `ping` network utility. ICMP is built on top of IP packets, so it is constrained by its maximum payload of 65,535 bytes minus headers. The general characteristics taking into account for data exfiltration are:

- ICMP protocol is **routable to Internet** as it is built on top of IP.
- ICMP protocol counts with **error detection** thanks to the `Checksum` field of the ICMP header.
- ICMP does not implement any type of reliability or replay mechanism, so **it is not assured that a packet will reach its destination**.

Echo and Echo Reply

These packets are used to test network connection between gateways or hosts. The echo message (Type 8) sends a `Data` field with any type of binary

data. An echo reply message (`Type` 0) **must reply** with the same data from echo message to fulfill standard specification. These types of packets are defined as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |     Code      |          Checksum             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Identifier          |        Sequence Number        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Data ...
+-+-+-+-+-
```

`Data` field is a variable length binary blob data, its size is 56 bytes when used legitimate for the network utility `ping`. Then, we can distinguish between two types of adversary: (1) a rough adversary will use up to 65,507 per packet, and (2) a stealthy adversary will try to impersonate legitimate `ping` traffic using just 56 bytes of data. Additionally, both of them can also use 2 byte `Identifier` and 2 byte `Sequence Number` field to exfiltrate data.

Destination Unreachable and Time Exceeded

Destination Unreachable (Type 3) is used by gateways to indicate that destination is unreachable, while Time Exceeded (Type 11) is sent by gateways to source when they encountered that a diagram has a time to live field equal to 0. Both packets follow the same structure, only changing the type field accordingly:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |     Code      |          Checksum             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             unused                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Internet Header + 64 bits of Original Data Datagram      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

In this case, both stealthy and rough adversaries can use the 4 bytes of the `unused` field to carry the exfiltrated data.

Parameter Problem

Parameter Problem (Type 12) is sent by gateway or host when an incoming datagram contains an error such as it has to be discarded (for example, the use of incorrect arguments in an option). This packet follows a similar structure than the previous packet:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |     Code      |          Checksum             |
```

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Pointer    |                  unused                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Internet Header + 64 bits of Original Data Datagram      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

As before, both stealthy and rough adversary can use the 3 bytes `unused` field to carry the exfiltrated data.

4.3.2 Network Time Protocol (NTP)

NTP is application-layer (layer 7 in OSI model) protocol used to synchronize system clocks among a set of distributed time servers and clients [40]. NTP is built on top of UDP transport-layer, so it is constrained by its maximum payload of 65,535 bytes minus headers. The general characteristics taking into account for data exfiltration are:

- NTP protocol is **routable to Internet** as it is built on top of UDP.
- NTP protocol counts with **error detection** thanks to the `Checksum` field of the UDP header.
- NTP does not implement any type of reliability or replay mechanism, so **it is not assured that a packet will reach its destination**.

NTP Format

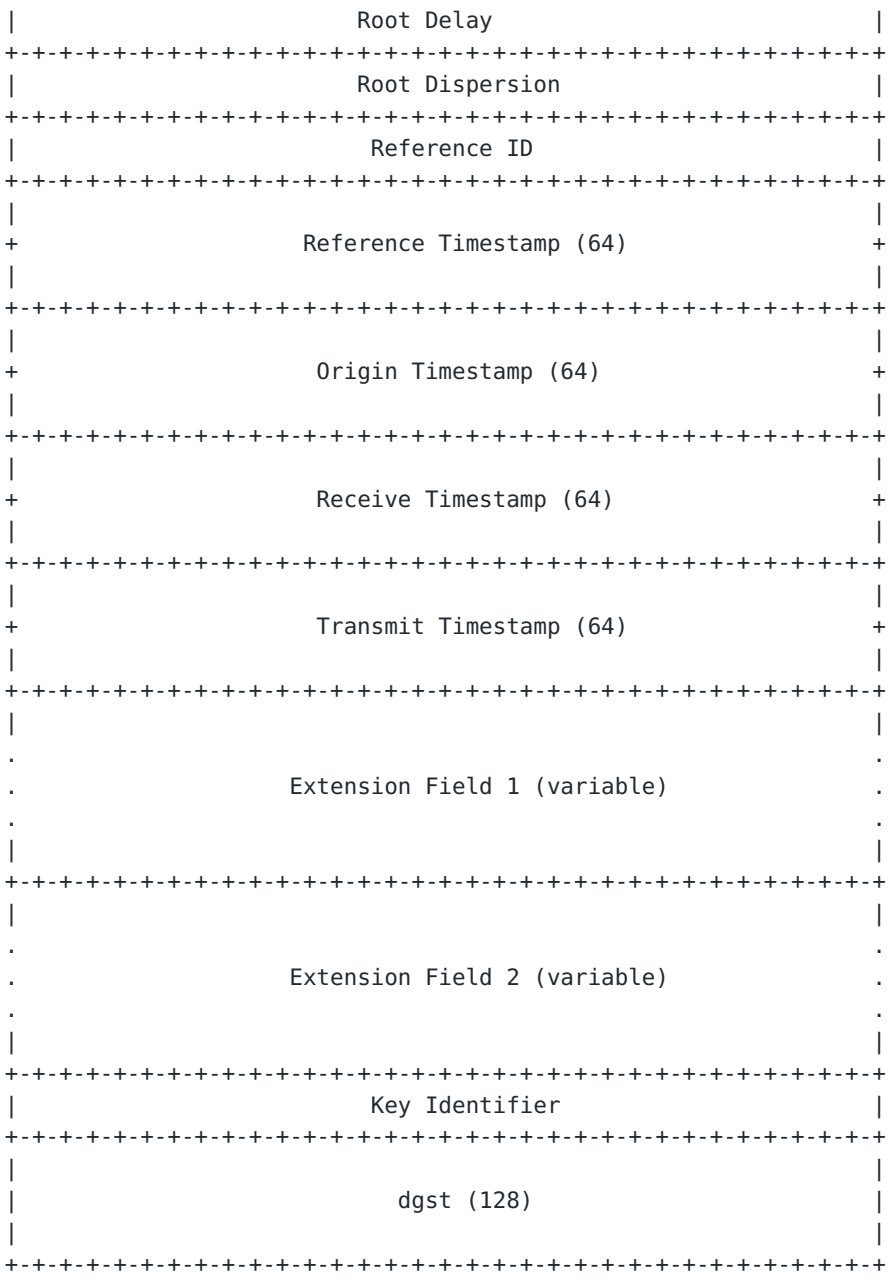
Both NTP client and server use one type of common packet format. Usually, a client requests the current time to a remote server, passing its own time into the request. The server adds four timestamps into the response packet:

- *Reference Timestamp*: the time when the system clock was last set or corrected.
- *Origin Timestamp*: the time at the client when the request departed for the server.
- *Receive Timestamp*: the time at the server when the request arrived from the client.
- *Transmit Timestamp*: the time at the server when the response left for the client.

Thanks to these timestamps, the client can derive the reference time from the server and the elapsed transmission time. This way, thanks to successive requests, the client can finally synchronize its clock with the NTP server time with an affordable time discrepancy.

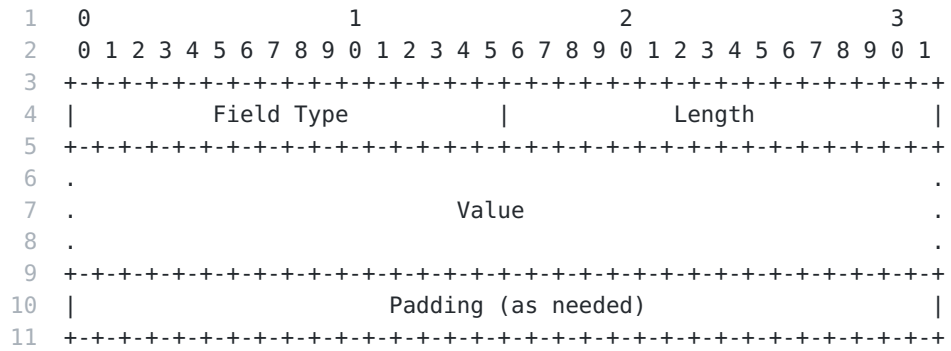
The NTP packet is defined as follows:

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|LI | VN  |Mode |    Stratum     |     Poll      |  Precision   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```



For this packet, both adversaries can use up to 42 bytes thanks to the sum of several fields: 1 byte in `Poll`, 1 byte in `Precision`, 4 bytes in `Root Delay`, 4 bytes in `Root Dispersion`, 8 bytes in `Reference Timestamp`, 8 bytes in `Origin Timestamp`, 8 bytes in `Receive Timestamp`, and 8 bytes in `Transmit Timestamp`.

Typical Windows and Linux utilities for NTP synchronization packets do not use neither extension fields nor MAC (key identifier plus digest) but, in current NTP version 4 [41], these fields are available to use. This means that a rough adversary could use at least one of the `Extension Field`, which is defined as:



If this `Extension Field` is present, a rough adversary can also add up to 65,418 bytes to the payload available into one single packet thanks to the sum of 2 bytes in Field Type and 65,416 bytes in Value Field (2 bytes are spent into specifying the `Lenght`). Additionally, the protocol specification states that if a `Extension Field` is present, it **must** be followed by a MAC field. That is, a rough adversary must also add a payload of 4 bytes in `Key Identifier` and 32 bytes in the `dgst`. In short, the final payload available for a rough adversary is 65,496 bytes.

4.3.3 Domain Name System (DNS)

DNS is a hierarchical and descentralized naming system responsible to translate domain names into IP addresses, either addresses corresponding to the Internet or inside a private network [42]. Additionally, DNS also designates the application-layer (layer 7 in OSI model) protocol used to communicate hosts with the name resolver hierarchy. The general characteristics of this protocol for data exfiltration are:

- DNS protocol is **routable to Internet** as it is built on top of UDP.
- DNS protocol counts with **error detection** thanks to the `Checksum` field of the UDP header.
- DNS does not implement any type of reliability or replay mechanism, so **it is not assured that a packet will reach its destination**.

The use of this protocol for data exfiltration requires to register a domain and register a root authority for that domain, so any DNS query to this domain can be processed by the adversary. All DNS queries are firstly constructed as the conjunction of DNS header plus the proper DNS question, and a response usually adds some other Resource Record (RR) to the server response (later discussed). This means that question packets can carry substantially less payload while server response can use RR to overcome this limitation. Therefore, data exfiltration upstream is much limited than its downstream counterpart.

While DNS protocol has a limitation of 65,535 corresponding to the Maximum Transmission Unit (MTU) of UDP, we consider that a stealthy adversary would adequate the total amount of DNS size to 1472 bytes, corresponding to the Ethernet MTU (1500 bytes - 20 IP header - 8 UDP header). Although a higher value is feasible in UDP, IP would *fragment* the UDP packet to fit Ethernet packets. So, a theoretical maximum UDP packet would be fragmented into

47 Ethernet packets, meaning that when only one of these fragments is lost, the whole UDP packet would be discarded.

The DNS header and DNS question present in **all** DNS queries and responses are as follows, respectively:

```
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      ID                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|QR|   Opcode  |AA|TC|RD|RA|   Z    |   RCODE   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    QDCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    ANCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    NSCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    ARCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

```
                                1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               |
/                     QNAME                     /
/                                               /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     QTYPE                     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     QCLASS                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

DNS Query

This technique uses information stored in the subdomains labels for data exfiltration [36]. These types of DNS queries use `QNAME` field to resolve a name similar to:

```
subdomain.subdomain.subdomain.domain.tld
```

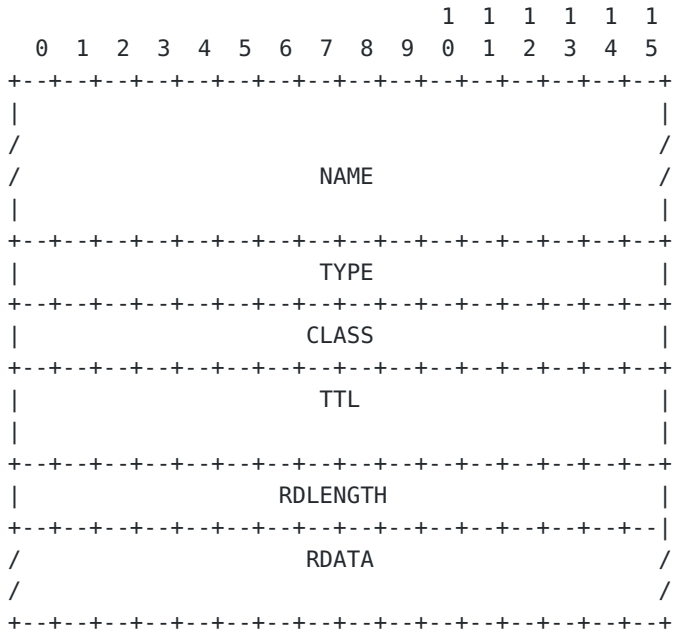
where each subdomain label has a limitation of 63 bytes as maximum size while the total length cannot exceed 255 bytes. Label encoding is built with a label length followed with the actual label and 0 is used to indicate the end of the name; so, a subdomain of the previous example is encoded as:

```
9|s|u|b|d|o|m|a|i|n|6|d|o|m|a|i|n|3|t|l|d|0
```

If domain and top-level domain (TLD) label are kept to a minimum, domain could has a size of 2 bytes (1 actual preceded by 1 byte length) and 3 bytes top-level domain (TLD) (2 bytes of nowadays minimum TLD preceded by 1 byte lenght). This combination keeps room for a total payload of 245 bytes, divided into a first 56-byte label and followed by three labels of 63-byte length each.

DNS Response

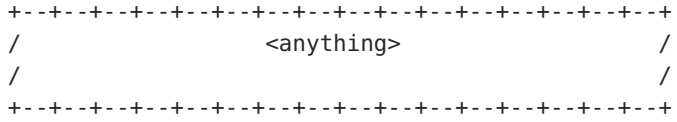
In the case of DNS response, it always contains the following Resource Record format appended to the query headers. The header defining the actual RR data is shown as `RDATA` field:



To maximize the RR payload, we limit the QNAME to just 6 bytes, 2 bytes for domain plus 3 bytes for TLD, plus 1 byte for the end of the stream.

NULL RDATA

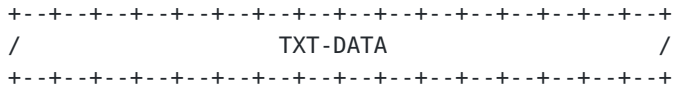
An arbritrary amount of binary data can be inserted in the NULL field until the UDP MTU is reached:



Therefore, a stealthy adversary can exfiltrate up to 1438 bytes bytes while a rough adversary can exfiltrate up to 65,473 bytes.

TXT RDATA

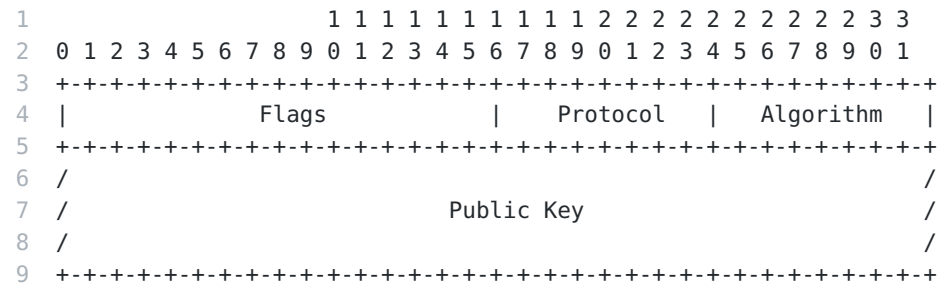
TXT-DATA contains one or more character-strings, and it is used to used to hold descriptive text. A character-string is a single length octet followed by the specified number of characters, treated as binary information, and can be up to 256 characters in length (including the length octet):



This construction limits the total payload to 65,217 bytes because we need to divide the data to exfiltrate into 254 chunks of 255 bytes (plus 1 byte to specify the length), and a remaining one of 199 bytes (plus 1 byte). Additionally, a stealthy adversary can exfitrate up to 1432 bytes following the same limitation.

DNSKEY RDATA

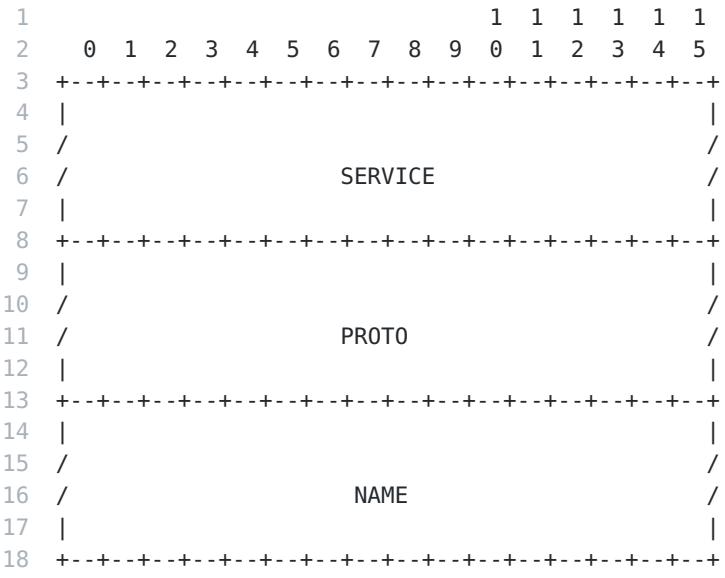
DNS Security Extensions (DNSSEC) are a collection of resource records and protocol modifications that provide source authentication for the DNS thanks to public key cryptography to sign and authenticate DNS RR [43]. In particular, the public keys are stored in DNSKEY RR and are used in the DNSSEC authentication process. DNSKEY RDATA is structured as follows:



An adversary can use the `Public Key` field to store the payload, but is **must be base64-encode** first. The Request for Comments (RFC) standard does not specify any maximum length limit for the public key, but we assume it is 512 bytes, equivalent to a 4096 bit RSA key.

SRV RDATA

The SRV RDATA allows to designate several servers for specified services in a single domain, thanks to the name and port contained in the record [44]. The SRV RDATA structure is as follows:



```
|                      TTL                      |
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     CLASS                     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                   PRIORITY                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    WEIGHT                     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     PORT                      |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               |
/                                               /
/                    TARGET                     /
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

In this case, an adversary can send up to 245 bytes in `TARGET` (following `QNAME` structure).

RRSIG RDATA

As part of DNSSEC authentication process, digital signatures are stored in RRSIG, which follows this structure:

```
                     1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Type Covered           |  Algorithm    |     Labels    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Original TTL                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Signature Expiration                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Signature Inception                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Key Tag           |                              /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+         Signer's Name        /
/                                                             /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                                                             /
/                          Signature                          /
/                                                             /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

In this case, an adversary can store up to 265 bytes into the payload thanks to the use of 245 bytes in `Signer's Name` (following `QNAME` structure), and 20 bytes in `Signature` (corresponding to the 160 bit RSA/SHA1 signature).

MX RDATA

An MX RR is responsible of specify the mail server corresponding a domain name. This RDATA is structured as follows:

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                  PREFERENCE                   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/                   EXCHANGE                    /
/                                               /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

An adversary can use in this case a useful payload of 247 bytes corresponding to 2 bytes in `PREFERENCE` field, and 245 bytes in `EXCHANGE` field (following `QNAME` structure).

CNAME RDATA

The CNAME RDATA is used to specify the canonical name of an alias. So, this record has the same payload limitation of 245 bytes as any other `QNAME`. The record's format on the wire is as follows:

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/                     CNAME                     /
/                                               /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

A RDATA

The A RDATA specifies the corresponding IPv4 Internet address associated to a given domain, so its payload can be up to 4 bytes:

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    ADDRESS                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

4.4 IoT Protocols

In this section, we discuss the protocols commonly used in IoT networks suitable to data exfiltration, either by IoT devices or IoT gateways. Namely, we focus on Constrained Application Protocol (CoAP), Message Queuing Telemetry Transport (MQTT), and Advanced Message Queuing Protocol (AMQP).

4.4.1 Constrained Application Protocol (CoAP)

The Constrained Application Protocol (CoAP) is a application-layer (layer 7 in OSI model) transfer protocol being used with constrained nodes and networks [45]. This protocol follows a request/response model quite similar to HTTP interaction model, where resources are accessed by a URI interface thanks to certain actions (verbs) on them. The general characteristics of this protocol for data exfiltration are:

- CoAP protocol is **routable to Internet** as it is built on top of UDP.
- CoAP protocol counts with **error detection** thanks to the `Checksum` field of the UDP header.
- CoAP does not implement any type of reliability or replay mechanism, so **it is not assured a the packet will reach its destination**.

While an UDP packet can carry a total payload of 65,535 bytes, the CoAP standard definition states that a IP MTU of 1280 bytes should be assumed. Thus, we distinguish between a stealthy adversary who adequates the packet size to the MTU and a rough adversary who maximizes it to the UDP limit. All CoAP packets follow the same structure, where `Code` field distinguishes between actions taking place into URIs:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Ver| T |  TKL  |      Code     |          Message ID           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Token (if any, TKL bytes) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Options (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 1 1 1 1 1 1|    Payload (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The `Option` field is built with the conjunction of an 4-bit `Option Delta`, which indicates the `Option` type; and a 4-bit `Option Length`. In the case of a `Option Length` bigger than 15 (`1111` bits), the special `13` length value is used to indicate that the following byte specifies the option length and, thus, it can be up to 255 bytes (`0xff` byte):

```
    0   1   2   3   4   5   6   7
+---------------+---------------+
|               |               |
|  Option Delta | Option Length |   1 byte
|               |               |
+---------------+---------------+
\                                \
/         Option Delta          /   0-2 bytes
\          (extended)           \
+-------------------------------+
\                                \
/         Option Length         /   0-2 bytes
\          (extended)           \
+-------------------------------+
\                                \
/                                /
\                                \
/         Option Value          /   0 or more bytes
\                                \
/                                /
```

```
\                              \
+------------------------------+
```

In particular, the CoAP options with their types are:

- `Uri-Host` specifies the Internet host of the resource being requested. This option is a UTF-8 string and it has a length ranging from 1 to 255 bytes.
- `Uri-Port` specifies the transport-layer port number of the resource. This option is an unsigned integer of size 0 to 2 bytes.
- `Uri-Path` specifies one segment of the absolute path to the resource. This option is a UTF-8 string and it has a length of 0 to 255 bytes, and a single CoAP packet can contain various paths.
- `Uri-Query` specifies one argument parameterizing the resource. This option is a UTF-8 string and it has a length of 0 to 255 bytes. A single CoAP packet can contain various queries.
- `Proxy-Uri` is used to make a request to a forward-proxy. This option is a UTF-8 string and it has a length of 1 to 1034 bytes.
- `Proxy-Scheme` is used to substitute an URI with the content of a Proxy-Scheme Option. This option is a UTF-8 string and it has a length of 1 to 255 bytes.
- `Content-Format` indicates the representation format of the message payload (e.g: `application/octet-stream`). This options is an unsigned integer of 0 to 2 bytes.
- `Accept` can be used to indicate which Content-Format is acceptable to the client. This option is an unsigned integer of 0 to 2 bytes.
- `Max-Age` indicates the maximum time a response can be cached before it is considered not fresh. This options is an unsigned integer of 0 to 4 bytes.
- `ETag` is intended for use as a resource-local identifier for differentiating between representations of the same resource that vary over time. An opaque sequence is a sequence of 1 to 8 bytes.
- `If-Match` is used to make a request conditional on the current existence or value of an ETag for one or more representations of the target resource. An opaque sequence is a sequence of 1 to 8 bytes.
- `If-None-Match` is used to make a request conditional on the non-existence of the target resource. This option has no value length.
- `Location-Path` along Location-Query option indicates a relative URI that consists either of an absolute path, a query string, or both. It can be used to indicate the location of the resource created as the result of a POST request. This option is a UTF-8 string and it has a length of 0 to 255 bytes.
- `Location-Query` is a UTF-8 string and it has a length of 0 to 255 bytes.
- `Size1` provides size information about the resource representation in a request. This options is an unsigned integer of 0 to 4 bytes.

GET and DELETE Methods

The GET method retrieves a representation for the information that currently corresponds to a resource identified by the requested URI, while the DELETE method requests to delete a resource identified by the request URI.

As both methods do not contain a CoAP payload, the data exfiltration mechanism relies on the use of the options. A stealthy adversary has several combinations to maintain the IP packet size under 1280 bytes MTU. For example, the stealthy adversary can use 2 bytes in `Message ID` field, and 8 bytes in `Token` field, which both are present in every CoAP message; plus 4 of every option containing a 255 bytes long UTF-8 string, and another one containing 210 bytes. This sum corresponds to a total payload of 1228 bytes.

While a rough adversary can use 2 bytes in `Message ID` field, 8 bytes in `Token` field, and 2594 bytes summed with one of every option except `If-None-Match`. Additionally, the adversary can add different `Uri-Path` and `Uri-Query` until the payload reaches the maximum of 64,993 bytes, corresponding to the addition of 244 options of 255 bytes plus another one of 169 bytes.

POST and PUT Methods

Both POST and PUT method indicate that the representation enclosed in the request creates or updates an specific resource, depending on the followed convection.

Both methods contain a CoAP payload to use it to exfiltrate data. As shown previously, there is a payload marker corresponding to the byte `0xf` after the options, which indicates the beginning of the payload. So, a stealthy adversary can exfiltrate up to 1245 bytes per packet thanks to 2 bytes in `Message ID` field, 8 bytes in `Token` field, 255 bytes in the mandatory `Uri-Path` option and 980 bytes in the proper payload, while 2 bytes are spent to indicate `application/octet-stream` content format. On the contrary, a rough adversary can exfiltrate 65,500 bytes in the payload, corresponding to 255 bytes in the `Uri-Path` and 65,245 in the proper payload.

4.4.2 Message Queuing Telemetry Transport (MQTT)

The Message Queuing Telemetry Transport (MQTT) is a application-layer (layer 7 in OSI model) implementing a client server publish/subscribe messaging transport protocol [46]. The general characteristics taking into account for data exfiltration are:

- MQTT protocol is **routable to Internet** as it is built on top of TCP/IP.
- MQTT protocol counts with **error detection** thanks to the `Checksum` field of the TCP header.
- Since MQTT is built on top of TCP, so **its delivery is assured**.

All packets in the MQTT 5.0 protocol follows the same structure:

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Fixed Header, present in all MQTT Control Packets   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
| Variable Header, present in some MQTT Control Packets |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Payload, present in some MQTT Control Packets     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The `Fixed Header` is included in all MQTT packets, but `Variable Header` and `Payload`, if present, are different for every packet type. In the case of the `Fixed Header`, it follows the following structure:

```
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|     CT    |     CF    |     RL (1-4 bytes)    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Where CT stands for Command Type, CF for Control Flag, and RL for Remaining Length of the packet. It should be noted that none of these fields can be used for data exfiltration.

In the case of the `Variable Header`, MQTT version 5.0 contains zero or more `Property` fields preceded of 1 to 4 bytes encoding the `Properties Length`. These properties were introduced in MQTT 5.0, but the `Variable Header` without properties and the `Payload` remains the same for older MQTT versions too. So, we contemplate a rough adversary which uses these properties, while a stealthy adversary is constrained to the `Payload` of the previous and stable MQTT 3.1.1 version and the `Variable Header` without properties. The available properties are:

- `Assigned Client Identifier` is a UTF-8 encoded string of a maximum length of 23 bytes.
- `Authentication Data` is a binary data up to 65,535 bytes.
- `Authentication Method` is a UTF-8 encoded string up to 65,535 bytes.
- `Content Type` is a UTF-8 encoded string up to 65,535 bytes.
- `Correlation Data` is a binary data up to 65,535 bytes.
- `Maximum Packet Size` is a four-byte integer.
- `Maximum QoS` is one byte.
- `Message Expiry Interval` is a four-byte integer.
- `Payload Format Indicator` is a byte which indicates that the payload is UTF-8 encoded character data.
- `Reason String` is a UTF-8 encoded string up to 65,535 bytes.
- `Receive Maximum` is a two-byte integer.
- `Request Problem Information` is one byte.
- `Request Response Information` is one byte.
- `Response Information` is a UTF-8 encoded string up to 65,535 bytes.
- `Response Topic` is a UTF-8 encoded string up to 65,535 bytes.
- `Retain Available` is one byte.
- `Server Keep Alive` is a two-byte integer.

- `Server Reference` is a UTF-8 encoded string up to 65,535 bytes.
- `Session Expiry Interval` is a four-byte integer.
- `Shared Subscription Available` is one byte.
- `Subscription Identifiers Available` is one byte.
- `Subscription Identifier` is a variable byte integer from 1 to 4 bytes.
- `Topic Alias Maximum` is a two-byte integer.
- `Topic Alias` is a two-byte integer.
- `User Property` is a UTF-8 string pair with a key up to 65,535 bytes and a value up to 65,535 bytes.
- `Wildcard Subscription` is one byte.

CONNECT

This is the first packet sent from the client to the server once they opened a network connection is opened. CONNECT packet counts with a variable header:

```
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|              Protocol Name Length             |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               |
/                 Protocol Name                 /
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|   Protocol Version    |     Connect Flags     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                  Keep Alive                   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|   Prop. L (1-4 bytes) |         Property      |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/         Property      |          ...          /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Where `Procotol Name` is "MQTT" and the payload for this packet consists in a unique `ClientID` field, corresponding to a UTF-8 encoded string of a maximum length of 23 bytes:

```
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/                   ClientID                    /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

So, a stealthy adversary can use 23 bytes for payload exfiltration while a rough adversary can add the `User Property`, `Authentication Method` and `Authentication Data` properties, rising the total payload up to 262,163 bytes.

CONNACK

The CONNACK packet is the packet sent by the server in response to a CONNECT packet received from a client. CONNACK variable header is as follows:

```
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|           CAF         |       Connect...      /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/    ...Reason Code     |  Prop. L (1-4 bytes)  /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/        Property       |        Property       /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/        Property       |          ...          /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Where `Connect Reason Code` `0x00` and there is no payload for this packet, so it is unsuitable to a stealthy adversary. However, a rough adversary can use up to 458,745 bytes adding up `Reason String`, `User Property`, `Response Information`, `Server Reference`, `Authentication Method`, and `Authentication Data`.

PUBLISH

A PUBLISH packet is sent from a client to a server or from a server to a client to transport an application message. PUBLISH variable header is as follows:

```
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/                  Topic Name                   /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|               Packet Identifier               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|   Prop. L (1-4 bytes) |         Property      |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/         Property      |          ...          /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Where a `Topic Name` is a UTF-8 encoded string up to 65,535 bytes, the `Packet Identifier` is a two byte integer, and the payload for this packet can be up to theoretical 268,435,456 bytes (approximately 256MiB). A stealthy adversary can limit the total payload to one IP packet total length minus headers, that is, 65,242 bytes into the payload and 255 bytes into the topic. On the contrary, a rough adversary can use the total amount of both fields to achieve a pyload of 268,500,991 bytes.

PUBACK, PUBREC, PUBREL and PUBCOMP

A PUBACK packet is the response to a PUBLISH packet with QoS 1, while PUBREC, PUBREL, and PUBCOMP are the packets interchanged between client and server to fulfill QoS 2. All these packets follow the same structure:

```
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| Packet Identifier MSB | Packet Identifier LSB |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|      Reason Code      |  Prop. L (1-4 bytes)  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/       Property        |          ...          /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Where `Reason Code 0x00` indicates success, and there is no payload for these types of packet. So, a stealthy packet is unable to use this packet for data exfiltration, while a rough adversary can use a total payload up to 196,605 bytes, thanks to adding `Reason String` and `User Property` properties.

SUBSCRIBE

The SUBSCRIBE packet is sent from the client to the server to create one or more subscriptions. SUBSCRIBE variable header follows this structure:

```
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|               Packet Identifier               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|   Prop. L (1-4 bytes) |         Property      |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/         Property      |          ...          /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Where `Packet Identifier` is a two-byte integer and the payload for the SUBSCRIBE packet is:

```
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                   Topic Length                |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/                   Topic Filter                /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  Subscription Options |          ...          |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Where the `Topic Filter` is a UTF-8 encoded string up to 65,535 bytes. So, a stealthy adversary can use this topic for payload up to 255, while a rough adversary can use all topic size plus the `User Property`, rising the payload up to 196,605 bytes.

SUBACK/UNSUBACK

A SUBACK packet is sent by the server to the client to confirm receipt and processing of a SUBSCRIBE packet. Likewise, UNSUBACK packet is sent to confirm receipt of an UNSUBSCRIBE packet. Both packet's variable header follows this structure:

```
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|               Packet Identifier               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|   Prop. L (1-4 bytes) |         Property      |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/         Property      |          ...          /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Where `Packet Identifier` is a two-byte integer, and the payload of this packet contains a list of `Reason Codes` (one byte each) corresponding to all subscribe topics in SUBSCRIBE packet. This packet is unsuitable to a stealthy adversary, while a rough adversary can use a total payload of 196,605 bytes thanks to `Reason String` and `User Property` properties.

UNSUBSCRIBE

An UNSUBSCRIBE packet is sent by the client to the server, to unsubscribe from topics. UNSUBSCRIBE variable header follows this structure:

```
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|               Packet Identifier               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|   Prop. L (1-4 bytes) |         Property      |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Where `Packet Identifier` and the payload for the UNSUBSCRIBE packet contains the list of topic filters from which the client wishes to unsubscribe:

```
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                  Topic Length                 |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/                  Topic Filter                 /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                       ...                     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

A stealthy adversary can use the `Topic Filter` for a payload up to 255, while a rough adversary can use all 65,535 bytes plus the `User Property` for a payload up to 196,605 bytes.

PINGREQ and PINGRESP

These ping-alike packets are sent from a client to the server to check aliveness. These packets do not have either variable header or payload, so they are not unsuitable to data exfiltration.

DISCONNECT

The DISCONNECT packet is the final MQTT control packet sent from the client or the server, and specifies the reason why the network connection is

being closed. DISCONNECT variable header follows this structure:

```
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|           DRC         |  Prop. L (1-4 bytes)  /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/        Property       |          ...          /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Where Disconnect Reason Code (`DRC`) is one byte long, and there is no payload for this type of packet. While a stealthy is unable to use these packet, a rough adversary can use a payload up to 262,176 bytes thanks to texttttReason String, `User Property` and `Server Reference` properties.

AUTH

An AUTH packet is sent as part of an extended authentication exchange, such as challenge/response authentication. AUTH variable header follows this structure:

```
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|           ARC         |  Prop. L (1-4 bytes)  /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/        Property       |          ...          /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

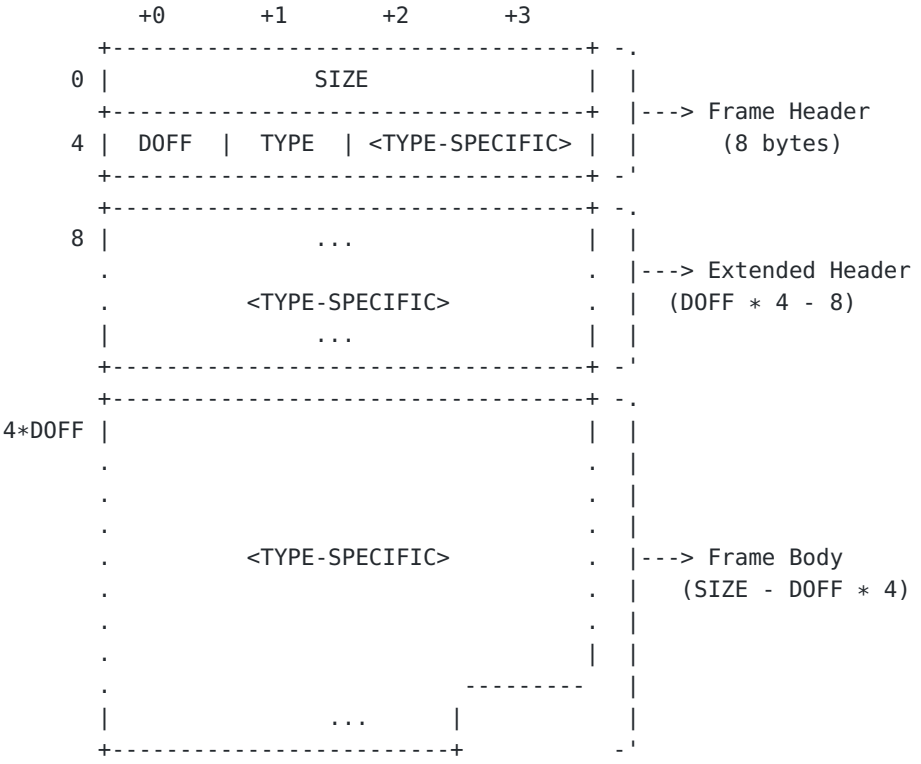
Where Authenticate Reason Code (`ARC`) is one byte long, and there is not payload for this packet. A rough adversary can exfiltrate a payload of 327,675 byte thanks to `Authentication Method`, `Authentication Data`, `Reason String` and `User Property` properties.

4.4.3 Advanced Message Queuing Protocol (AMQP)

The Advanced Message Queuing Protocol (AMQP) 1.0 is a application-layer (layer 7 in OSI model) for message-oriented middleware [47]. One of its main characteristics is that it defines a binary wire-level peer-to-peer protocol for transporting messages between two processes over a network. An AMQP-encoded data stream consists of untyped bytes with embedded constructors, which indicates how to interpret the untyped bytes that follows. As the binary encoding is not ideal to represent types, AMQP follows a XML schema for the type notation. The general characteristics of the protocol taking into account for data exfiltration are:

- AMQP protocol is **routable to Internet** as it is built on top of TCP/IP.
- AMQP protocol counts with **error detection** thanks to the `Checksum` field of the TCP header.
- AMQP is built on top of TCP, so **its delivery is assured**.

All packets of AMQP follow the same structure: (1) a `Frame Header` of fixed 8 bytes size structure that precedes each frame, (2) a variable `Extended Header` for future expansion, and (3) a variable `Frame Body` containing a sequence of bytes whose format depends on the frame type:



AMQP is a complex protocol which counts on more than 24 native data types, plus 19 specific definitions built on those native date types. Therefore, only those used for the later described AMQP packet types are listed below:

- $*$: it can represent any AMQP type.
- `boolean`, `role`: 1 byte representing true or false.
- `delivery-number`, `delivery-tag`: up to 32 bytes of binary data.
- `symbol`, `ietf-language-tag`: up to 4,294,967,295 bytes ($2^{31} - 1$ or approximately 4GiB) representing ASCII characters.
- `map`, `fields`: key/value dictionary up to 4GiB.
- `uint`, `milliseconds`, `transfer-number`, `handle`, `sequence-no`, `message-format`: 4 bytes representing an unsigned integer.
- `receiver-settle-mode`: 1 byte restricted to the 0-1 range.
- `sender-settle-mode`: 1 byte restricted to the 0-2 range.
- `string`: up to 4GiB representing UTF-8 characters.
- `ulong`: 8 bytes representing an unsigned long integer.
- `ushort`: 2 bytes representing an unsigned short integer.
- `error`: a complex data type compounded of three types: `symbol`, `string`, `fields`.

In the case of types similar to a `string`, we considered that a stealthy adversary can use a size of `PATH_MAX`[2] (4096 bytes), as defined in Linux. Despite that some data types can be up to 4GiB, they are restricted to the maximum frame size established by the four-byte `SIZE` field of the `Frame Header`. This restrictions means that **the maximum frame size is 4GiB** minus headers and, thus, a frame cannot hold two data types to their maximum size. Even so, we appoint the theoretical maximum payload of each packet if all data types are added.

In the following subsections, each AMQP packet type corresponding to one AMQP action (called performative) is discussed and shown in its XML notation. In this case, we consider that a stealthy adversary uses the mandatory fields for every performative, while a rough a adversary uses all of them.

Open

The Open performative negotiates the connection parameters. This performative follows the XML schema:

```
1  <type name="open" class="composite" source="list" provides="frame">
2      <descriptor name="amqp:open:list" code="0x00000000:0x00000010"/
           >
3      <field name="container-id" type="string" mandatory="true"/>
4      <field name="hostname" type="string"/>
5      <field name="max-frame-size" type="uint" default="4294967295"/>
6      <field name="channel-max" type="ushort" default="65535"/>
7      <field name="idle-time-out" type="milliseconds"/>
8      <field name="outgoing-locales" type="ietf-language-tag"
           multiple="true"/>
9      <field name="incoming-locales" type="ietf-language-tag"
           multiple="true"/>
10     <field name="offered-capabilities" type="symbol" multiple="true
           "/>
11     <field name="desired-capabilities" type="symbol" multiple="true
           "/>
12     <field name="properties" type="fields"/>
13 </type>
```

A stealthy adversary can use `container-id` to exfiltrate up to 4096 bytes, while a rough adversary can use `container-id`, `hostname`, `outgoing-locales`, `incoming-locales`, `offered-capabilities`, `desired-capabilities`, `properties` fields to exfiltrate up to 28GiB.

Begin

The Begin performative begins a session on a channel. This performative follows the XML schema:

```
1  <type name="begin" class="composite" source="list" provides="frame"
       >
```

[2]Linux Kernel: https://github.com/torvalds/linux/blob/master/include/uapi/linux/limits.h (visited on 2020/09/09)

```
    <descriptor name="amqp:begin:list" code="0x00000000:0x00000011"
        />
    <field name="remote-channel" type="ushort"/>
    <field name="next-outgoing-id" type="transfer-number" mandatory
        ="true"/>
    <field name="incoming-window" type="uint" mandatory="true"/>
    <field name="outgoing-window" type="uint" mandatory="true"/>
    <field name="handle-max" type="handle" default="4294967295"/>
    <field name="offered-capabilities" type="symbol" multiple="true
        "/>
    <field name="desired-capabilities" type="symbol" multiple="true
        "/>
    <field name="properties" type="fields"/>
</type>
```

A stealthy adversary can use a payload of 12 bytes, distributed in 4 bytes in `next-outgoing-id`, 4 bytes in `incoming-window`, 4 bytes in `outgoing-windows`. On the contrary, a rough adversary can maximize the payload to be up to 12 GiB thanks to the use of `symbol` and `fields` types, holding 4GiB each.

Attach

The Attach performative attaches a link to a session. This performative follows the XML schema:

```
<type name="attach" class="composite" source="list" provides="frame
    ">
    <descriptor name="amqp:attach:list" code="0x00000000:0x00000012
        "/>
    <field name="name" type="string" mandatory="true"/>
    <field name="handle" type="handle" mandatory="true"/>
    <field name="role" type="role" mandatory="true"/>
    <field name="snd-settle-mode" type="sender-settle-mode" default
        ="mixed"/>
    <field name="rcv-settle-mode" type="receiver-settle-mode"
        default="first"/>
    <field name="source" type="*" requires="source"/>
    <field name="target" type="*" requires="target"/>
    <field name="unsettled" type="map"/>
    <field name="incomplete-unsettled" type="boolean" default="
        false"/>
    <field name="initial-delivery-count" type="sequence-no"/>
    <field name="max-message-size" type="ulong"/>
    <field name="offered-capabilities" type="symbol" multiple="true
        "/>
    <field name="desired-capabilities" type="symbol" multiple="true
        "/>
    <field name="properties" type="fields"/>
</type>
```

A stealthy adversary can use a payload of 4100 bytes, distributed in 4096 bytes in `name`, and 4 in `handle`. Likewise, a rough adversary can maximize the

payload to be up to 28 GiB thanks to the maximum size of types `string`, `map`, `symbol`, `fields` and using `string` in wildcard types (`*`).

Flow

The Flow performative updates the flow state for the specified link. This performative follows the XML schema:

```
<type name="flow" class="composite" source="list" provides="frame">
    <descriptor name="amqp:flow:list" code="0x00000000:0x00000013"/
        >
    <field name="next-incoming-id" type="transfer-number"/>
    <field name="incoming-window" type="uint" mandatory="true"/>
    <field name="next-outgoing-id" type="transfer-number" mandatory
        ="true"/>
    <field name="outgoing-window" type="uint" mandatory="true"/>
    <field name="handle" type="handle"/>
    <field name="delivery-count" type="sequence-no"/>
    <field name="link-credit" type="uint"/>
    <field name="available" type="uint"/>
    <field name="drain" type="boolean" default="false"/>
    <field name="echo" type="boolean" default="false"/>
    <field name="properties" type="fields"/>
</type>
```

A stealthy adversary can use a payload of 12 bytes, distributed in 4 bytes in `incoming-window`, 4 in `next-outgoing-id`, and 4 bytes in `outgoing-window`. On the contrary, a rough adversary can maximize the payload to be up to 4 GiB thanks to the `properties` field.

Transfer

The Transfer performative sends messages across a link. This performative follows the XML scheme:

```
<type name="transfer" class="composite" source="list" provides="
    frame">
    <descriptor name="amqp:transfer:list" code="0
        x00000000:0x00000014"/>
    <field name="handle" type="handle" mandatory="true"/>
    <field name="delivery-id" type="delivery-number"/>
    <field name="delivery-tag" type="delivery-tag"/>
    <field name="message-format" type="message-format"/>
    <field name="settled" type="boolean"/>
    <field name="more" type="boolean" default="false"/>
    <field name="rcv-settle-mode" type="receiver-settle-mode"/>
    <field name="state" type="*" requires="delivery-state"/>
    <field name="resume" type="boolean" default="false"/>
    <field name="aborted" type="boolean" default="false"/>
    <field name="batchable" type="boolean" default="false"/>
</type>
```

A stealthy adversary can use a payload of 4 bytes in `handle`, while a rough adversary can maximize the payload to be up to 4 GiB thanks to using the `state` field into any other type supporting such size. Additionally, both adversaries can also use the message payload. A stealthy can add a payload up to 65,443 till the maximum IP packet is reached (65,535 bytes). On the contrary, a rough adversary can use it to transfer up to 4GiB of payload.

Disposition

The Disposition informs the remote peer of local changes in the state of deliveries. This performative follows the XML scheme:

```
<type name="disposition" class="composite" source="list" provides="frame">
    <descriptor name="amqp:disposition:list" code="0x00000000:0x00000015"/>
    <field name="role" type="role" mandatory="true"/>
    <field name="first" type="delivery-number" mandatory="true"/>
    <field name="last" type="delivery-number"/>
    <field name="settled" type="boolean" default="false"/>
    <field name="state" type="*" requires="delivery-state"/>
    <field name="batchable" type="boolean" default="false"/>
</type>
```

A stealthy adversary can use a payload of 32 bytes in`first`, while a rough adversary can maximize the payload to be up to 4 GiB thanks to using the `state` field into any other type supporting such size.

Detach

The Detach performative detachs the link endpoint from the session. This performative follows the XML scheme:

```
<type name="detach" class="composite" source="list" provides="frame">
    <descriptor name="amqp:detach:list" code="0x00000000:0x00000016"/>
    <field name="handle" type="handle" mandatory="true"/>
    <field name="closed" type="boolean" default="false"/>
    <field name="error" type="error"/>
</type>
```

A stealthy adversary can use a payload of 4 bytes in`handle`. In this case, only using `error` field, a rough adversary can exfiltrate up to 12GiB.

End

The End perfomative indicates that the session has ended. This performative follows the XML scheme:

```
<type name="end" class="composite" source="list" provides="frame">
    <descriptor name="amqp:end:list" code="0x00000000:0x00000017"/>
    <field name="error" type="error"/>
```

```
4 </type>
```

In this packet type, there is no mandatory fields, so a stealthy cannot exfiltrate any data. On the contrary, a rough adversary can exfiltrate up to 12GiB thanks to the `error` field.

Close

The Close performative indicates that the sender will not be sending any more frames on the connection. This performative follows the XML scheme:

```
1 <type name="close" class="composite" source="list" provides="frame"
    >
2     <descriptor name="amqp:close:list" code="0x00000000:0x00000018"
        />
3     <field name="error" type="error"/>
4 </type>
```

As the previous performative, there is no mandatory fields to be used for a stealthy adversary. However, a rough adversary can exfiltrate up to 12GiB thanks to the `error` field.

4.5 Comparative

Table 4.1 resumes the general characteristics of all protocols. Moreover, a comparison of traditional protocols is summarized in Table 4.2, while IoT protocols are shown in Table 4.3. Both tables represent the transmission of 1MiB of data and they are divided in two main columns, one column for a stealthy adversary and another column for a rough adversary.

For each adversary, *Packet Size* represents the size of **one** packet to encapsulate the packet payload (roughly speaking, payload plus packet headers), *Overhead* is the **total** amount of payload transmitted divided by the size of the total packets sent, and *Nº of packets* represents the **total** number of packets necessary to sent the data.

Table 4.1: Summary of general characteristics by protocol.

	Transport Layer	Routable	Error Detection	Assured Delivery
ICMP	-	✓	✓	✗
NTP	UDP	✓	✓	✗
DNS	UDP	✓	✓	✗
CoAP	UDP	✓	✓	✗
MQTT	TCP	✓	✓	✓
AMQP	TCP	✓	✓	✓

Table 4.2: Exfiltration of 1,048,576 bytes (1 MiB) by traditional protocol.

	Stealthy Adversary			Rough Adversary		
	Nº of packets	Overhead	Packet Size	Nº of packets	Overhead	Packet Size
ICMP						
Echo/Echo Reply	18,725	14.29%	64	17	0.04%	65,515
Dest. Unreachable/Time Exceeded	262,144	800%	36	262,144	800%	36
Parameter Problem	349,526	1100%	36	349,526	1100%	36
NTP						
NTP Packet	24,967	14.29%	48	17	0.01%	65,504
DNS						
DNS Query	4280	10.64%	271	4,280	10.64%	271
NULL RDATA	730	2.37%	1472	17	0.10%	65,507
TXT RDATA	733	2.80%	1472	17	0.49%	65,507
DNSKEY RDATA	2048	7.03%	548	2048	7.03%	548
SRV RDATA	4280	28.60%	315	4280	28.60%	315
RRSIG RDAT	3957	23.40%	327	3957	23.40%	327
MX RDATA	4246	16.22%	287	4246	16.22%	287
CNAME RDATA	4280	16.36%	285	4280	16.36%	285
A RDATA	262,144	850%	38	262,144	850%	38

Table 4.3: Exfiltration of 1,048,576 bytes (1 MiB) by IoT protocol.

	Stealthy Adversary			Rough Adversary		
	Nº of packets	Overhead	Packet Size	Nº of packets	Overhead	Packet Size
CoAP						
GET/DELETE	853	4.06%	1280	17	4.84%	65,507
POST/PUT	850	3.64%	1280	17	0.05%	65,507
MQTT		(*3.1.1*)			(*5.0*)	
CONNECT	45,591	52.18%	35	4	< 0.01%	262,189
CONNACK	-	-	-	3	< 0.01%	458,775
PUBLISH	17	< 0.01%	65,507	1	< 0.01%	1,048,584
PUBACK/PUBREC/PUBREL/PUBCOMP	-	-	-	6	< 0.01%	196,622
SUBSCRIBE	4113	3.14%	263	6	< 0.01%	196,622
SUBACK/UNSUBACK	-	-	-	6	< 0.01%	196,622
UNSUBSCRIBE	4113	2.75%	262	6	< 0.01%	196,620
PINGREQ/PINGRESP	-	-	-	-	-	-
DISCONNECT	-	-	-	4	< 0.01%	262,176
AUTH	-	-	-	4	< 0.01%	327,696
AMQP						
Open	256	0.37%	4111	1	< 0.01%	1,048,591
Begin	87,382	133.34%	28	1	< 0.01%	1,048,594
Attach	256	0.54%	4118	1	< 0.01%	1,048,593
Flow	87,382	133.34%	28	1	< 0.01%	1,048,594
Transfer	17	0.08%	65,495	1	< 0.01%	1,048,590
Disposition	32,768	46.48%	47	1	< 0.01%	1,048,593
Detach	262,144	350%	18	1	< 0.01%	1,048,594
End	-	-	-	1	< 0.01%	1,048,593
Close	-	-	-	1	< 0.01%	1,048,593

Chapter 5

Chiton Library

Once we have studied all IoT protocols, we aim at developing a library to exfiltrate data encapsulating the data into IoT protocol's packets. This library is later used to measure the exfiltration time for different sizes of data.

5.1 Analysis

We have established the following functional and non-functional requirements. Recall that a functional requirement (RF) defines the required functionalities of the system, while a non-functional requirement (NFR) describes desirable characteristics such portability, reliability, etc.

Functional Requirements

FR1 The user shall encapsulate binary data into a protocol packet.

FR2 The user shall send protocol packets to another host over Internet.

FR3 The user shall de-encapsulate binary data from a protocol packet.

FR4 The user shall receive protocol packets coming from Internet.

FR5 The user shall be able to choose between different types of packets in a protocol.

FR6 The user shall specify addresses and ports during transmission.

FR7 The user shall specify the maximum payload that a packet can encapsulate.

Non-Functional Requirements

NFR1 The library shall run in Windows, macOS, and Linux.

NFR2 The library shall be extensible to support new protocols.

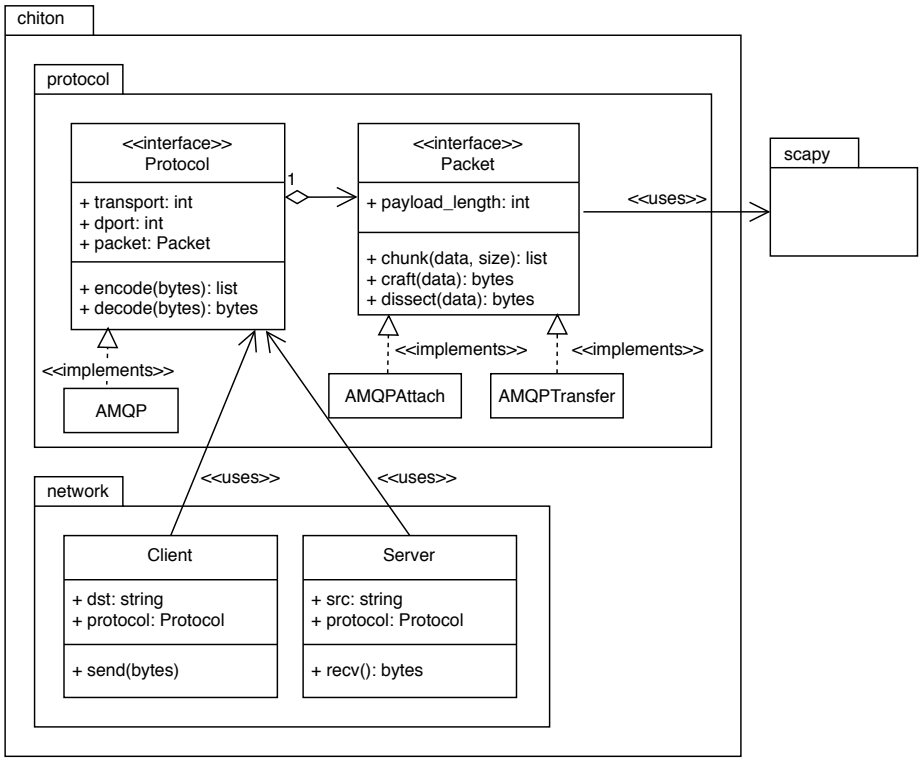


Figure 5.1: Class diagram of `chiton` library.

5.2 Design

In order to support multiple platforms (NFR1), this library was developed in Python programming language. Python is a multiplatform, high-level programming language that allows fast prototyping. Additionally, Python counts with a well-established and mature packet manipulation library called Scapy.

Scapy is a command-line interface (CLI) and library that allows you to manipulate network packets, automatically convert them to the on-wire bytes representation, and convert bytes back to packets objects. Additionally, Scapy also supports protocol extension (NFR2). Scapy already supports CoAP and MQTT, but not AMQP. In this work, we have extended Scapy to support AMQP as well.

The solution to meet the functional and non-functional requirements is shown in the Figure 5.1. The relevant aspects are:

- **Client**: this class receives a `Protocol`, which will be used to encode a buffer and `send` it over Internet to the `dst`.
- **Server**: this class is the `Client` counterpart, which also receives a `Protocol` to decode `recv` bytes over Internet from a `src`.
- **Protocol**: this interface specifies the `decode` and `encode` operations that every protocol must implement. With the use of this interface, details from a specific protocol are hidden to either the client or the server. Ad-

ditionally, this interface allows to transparently extend the library with new protocols.

- **Packet**: this interface specifies the `craft` and `dissect` operations to be implemented. Each `Packet` implementation needs to deal with the protocol packet specifics to `craft` and `dissect` the desired payload. We call `craft` to the encapsulation of the data to exfiltrate into the fields of a packet and, on the contrary, `dissect` is the de-encapsulation of the the data residing in the fields of a packet.

5.3 Example of Usage

The usage of the library is straightforward. In order to communicate two host, the client should be placed in the host with the data to exfiltrate while the server should be placed in the host willing to receive the exfiltrated data. Each pair of the communication must previously agree on the protocol and packet type used for data transfer.

An example script of a client is shown in Listing 5.1. In this example, the file to exfiltrate is passed as argument to the script, which reads it as binary data (line 12). The chosen protocol in this case is MQTT with the packet type PUBLISH (line 7), and this protocol is used to create a client object willing to communicate with the host at address 192.168.1.40 (line 8). Finally, the data to exfiltrate is simply sent to the desired host calling the `send` function with the data as parameter, and the library handles the encapsulation of the data into the protocol's packet payload (Figure 5.2).

Listing 5.1: `Chiton` client example.

```
1  import sys
2
3  import chiton.network.client as client
4  import chiton.protocol.mqtt as mqtt
5
6  def main(filename):
7      protocol = mqtt.MQTT(mqtt.PUBLISH)
8      mqtt_client = client.Client(protocol, '192.168.1.40')
9      data = read_file(filename)
10     mqtt_client.send(data)
11
12 def read_file(filename):
13     with open(filename, 'rb') as f:
14         return f.read()
15
16 if __name__ == '__main__':
17     main(sys.argv[1])
```

In the other side of the communication, the example server script should be running (Listing 5.2). The workflow is similar to the client, as the server needs to specify the protocol used for data transfer (line 7), and data is finally received (line 9). The library automatically de-encapsulates the packet payload to reassemble the original payload, ready to be saved into a file (line 10).

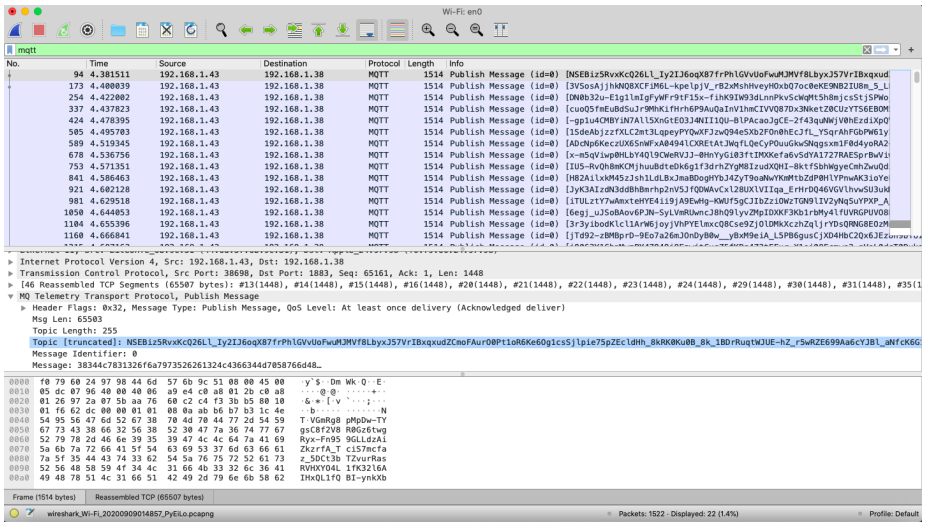


Figure 5.2: Data encapsulation into available payload of MQTT PUBLISH.

Listing 5.2: `Chiton` server example.

```
1  import sys
2
3  import chiton.network.server as server
4  import chiton.protocol.mqtt as mqtt
5
6  def main():
7      protocol = mqtt.MQTT(mqtt.PUBLISH)
8      mqtt_server = server.Server(protocol)
9      data = mqtt_server.recv()
10     write_file(data, 'output.bin')
11
12 def write_file(data, filename):
13     with open(filename, 'wb') as f:
14         f.write(data)
15
16 if __name__ == '__main__':
17     main()
```

Chapter 6

Experimentation

The goal of the experimentation is to empirically measure the elapsed time needed for data exfiltration though the studied IoT protocols.

6.1 Experiment Settings

In this experiment scenario, we simulated a basic adversary model. We tested how IoT protocols perform for different types of data varying from 1, 10, 100, 1000, 10000 to 100000 KiBs. This progression allows to compare the IoT protocol effectiveness for data exfiltration when the data file size increases. Since the nature of the data file is irrelevant, we generated a random test file with the maximum payload (100 MiB) and split it up to into chunks of the desired size. The random file was generated with the following command:

```
1 $ head -c 104857600 < /dev/urandom > test-file.bin
```

In order to check file integrity, we compare the SHA256 hash of the every chunk in the client and server, which both of them should be equal. The hash is calculated as follows in a Linux system:

```
1 $ sha256sum test-file.bin
2 a62aee7e5afdf6c6b49fb14116edb7c9dc3c94275fbf7ec4e707680569ca66df
    test-file.bin
```

The elapsed time is measured in the server side. As previously mentioned, the server is responsible of receiving the exfiltrated data. The first timestamp is generated when a new packet arrives, and the second timesptamp is generated when the last packet arrives. Consequently, the elapsed time is calculated subtracting the first timestamp to the second one. In this measurement, the network latency plays a big role into the data exfiltration time so, to overcome this issue, we measured the average of five different measurements taken during one day long.

We tested the best suitable type of packet for each protocol to exfiltrate data, that is, the one with the less number of packets and the less overhead as shown in Table 4.3. In particular, we tested CoAP POST/PUT, MQTT PUBLISH, and AMQP Transfer.

In order to execute the experiments and simulate a real case scenario, two hosts are connected over the Internet. The distance between hosts is 14 hops,

as measured by `traceroute`, and there is an average latency of 21.546ms, as measured by the `ping` utility.

- The first host is a MacBook Air, which plays the role of a compromised computer exfiltrating data. The computer components are a Intel i7-5650U at 2.2GHz, 8GB of DDR3 RAM at 1600 MHz, and a 802.11ac Wi-Fi.
- The second host is a personal computer (PC), which plays the role of the adversary receiving the exfiltrated data. This computers owns a Intel i7-6700 at 3.40GHz, 32GB of DDR4 RAM at 2133 MHz, and a 1000 Base I219-LM Ethernet connection.

In this scenario, the MacBook is executing a `chiton`'s client instance and is responsible to exfiltrate all the files, while the PC is executing the `chiton`'s server to receive all the files over the network. The PC also executes the process `tcpdump` to save a network trace in case of necessity.

6.2 Results and Discussion

The results are presented in Figure 6.1, where a big difference between the data exfiltration time from CoAP protocol to the MQTT and AMQP protocols can be spotted. We can interpret three main reasons for these results:

- *CoAP protocol needs to send more packets for the same amount of data*. This behavior implies more time overhead since the OS, in conjunction to the network device, must deal with the underlying I/O network more frequently.
- For the CoAP protocol, we kept IP packets under the 1280 bytes length limit, as stated in the Section 4.4.1. However, *there is a lack of knowledge about the MTU used in the network*. Thus, despite using the upper-bound limit established in the CoAP standard specification, UDP does not count with any mechanism to recover the MTU in use, and UDP packets can be bigger than the actual MTU. This limitation of the transport layer implies, for example, that an intermediate node with a smaller MTU needs to fragment UDP packets during transmission while the final end hostnode is responsible to reassemble these fragments, consuming more time. This specific problem is addressed with the use of TCP/IP, as in AMQP and MQTT, thanks to Path MTU Discovery mechanisms.
- Depending of how the intermediate nodes are configured, *they may apply more priority to TCP traffic over UDP*. Hence, UDP traffic could be being buffered by intermediate nodes and thus arrive later.

Additionally, we did not encounter any problem about UDP data loss or disorder in our experiments. However, it should be noted that UDP is a best-effort transport layer, meaning that on-transit packets are likely to be dropped or disordered for infinity of reasons. Therefore, the client needs to resend all the data in case of packet loss, drastically increasing the exfiltration time. Currently, `chiton` library can detect disorder or loss of UDP packets, but does not count with any replay packet mechanism. As future work, we aim to incorporate a replay packet mechanism to `chiton` to overcome these issues.

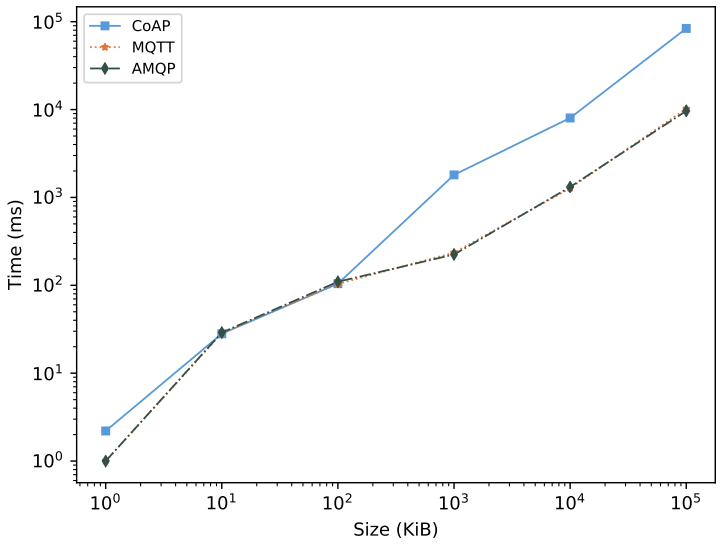


Figure 6.1: Comparison of data exfiltration times by IoT protocol.

Chapter 7

Conclusions and Future Work

With the increasing use of IoT devices and the concern about the privacy invasion, understanding how IoT devices can be abused is a relevant topic.

This work studies CoAP, MQTT, and AMQP IoT protocols from the point of view of data exfiltration. Data exfiltration is the unauthorized transfer of information from an information system. Additionally, we also contribute with the creation of `chiton`, a Python library that allows to exfiltrate data through the aforementioned three protocols.

While there are works focusing in how the protocols can be use to transmit data; to the best of our knowledge, this is the first study to extensively compare these IoT protocols from the point of view of data exfiltration, focusing on characteristics such as overhead and useful payload for every available packet.

Additionally, we empirically measure and compare the time necessary to exfiltrate files of different data size. The results show that CoAP is the less suitable protocol for data exfiltration, and being outperformed by both MQTT and AMQP. This preference is also supported from the point of view of an adversary, since this protocols are usually used to connect enterprise IoT networks to IoT cloud providers, so they are more likely to be allowed in the network of the organization. In this matter, MQTT is the protocol supported for the three major cloud providers (Amazon Web Services, Microsoft Azure, and Google Cloud).

As a future work, we propose to extend the scope of this study with the addition of more protocols and the measurement of the performance impact in the IoT devices. Regarding `chiton`, we aim to add support for a reply mechanism for lost UDP packets and for packet ordination, as well as to extend the number of supported protocols. Moreover, this study can be taken as the basis to study how to detect an unintended data transfer using IoT protocols, and thus improving detection systems.

Glossary

Covert channel Any communication channel that can be exploited by a process to transfer information in a manner that violates the systems security policy.

Data exfiltration The unauthorized transfer of information from an information system.

Data Loss Prevention (DLP) System that ensures that sensitive data is not lost, misused, or accessed by unauthorized users.

Internet of Things (IoT) Integration of various sensors, objects and smart nodes that are capable of communicating with each other without human intervention.

Side channel Leaked information in an unintended way as a result of computer activity.

Steganography The art of hiding a subliminal message into another medium.

Tunneling Tunnel one protocol into another.

Bibliography

[1] McAfee, "Grand Theft Data II: The Drivers and Shifting State of Data Breaches," 2019.

[2] NIST, *Glossary: Exfiltration*. URL: https://csrc.nist.gov/glossary/term/exfiltration (visited on 2020-05-21).

[3] B. W. Lampson, "A Note on the Confinement Problem," *Commun. ACM*, vol. 16, p. 613–615, Oct. 1973.

[4] U.S. Department Of Defense, *Trusted Computer System Evaluation Criteria*, pp. 1–129. London: Palgrave Macmillan UK, 1985.

[5] K. Xu, P. Butler, S. Saha, and D. Yao, "DNS for Massive-Scale Command and Control," *IEEE Transactions on Dependable and Secure Computing*, vol. 10, no. 3, pp. 143–153, 2013.

[6] M. Conti, A. Dehghantanha, K. Franke, and S. Watson, "Internet of Things security and forensics: Challenges and opportunities," *Future Generation Computer Systems*, vol. 78, pp. 544 – 546, 2018.

[7] IoT Analytics, *State of the IoT 2018: Number of IoT devices now at 7B – Market accelerating*. URL: https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/ (visited on 2020-05-21).

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

[9] C. H. Rowland, "Covert channels in the TCP/IP protocol suite," *First Monday*, vol. 2, no. 5, 1997.

[10] S. Zander, G. Armitage, and P. Branch, "A Survey of Covert Channels and Cuntermeasures in Computer Network Protocols," *IEEE Communications Surveys & Tutorials*, vol. 9, no. 3, pp. 44–57, 2007.

[11] I. S. Moskowitz, S. Russell, and B. Jalaian, "Steganographic Internet of Things: Graph Topology Timing Channels," in *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[12] A. Mileva, *Steganography in the World of IoT*, 2018. URL: http://eprints.ugd.edu.mk/20424/1/SECFOR.pdf (visited on 2020-05-21).

[13] A. Mileva, A. Velinov, and D. Stojanov, "New covert channels in Internet of Things," in *The 12th International Conference on Emerging Security Information, Systems and Technologies*, 2018.

[14] W. Bender, D. Gruhl, N. Morimoto, and A. Lu, "Techniques for data hiding," *IBM Systems Journal*, vol. 35, no. 3.4, pp. 313–336, 1996.

[15] G. J. Simmons, "The Prisoners' Problem and the Subliminal Channel," in *Advances in Cryptology*, pp. 51–67, Springer, 1984.

[16] C. Girling, "Covert Channels in LAN's," *IEEE Transactions on Software Engineering*, vol. 13, pp. 292–296, feb 1987.

[17] S. Craver, "On public-key steganography in the presence of an active warden," in *Information Hiding* (D. Aucsmith, ed.), (Berlin, Heidelberg), pp. 355–368, Springer Berlin Heidelberg, 1998.

[18] daemon9, "Project Loki," *Phrack Magazine*, vol. 7, August 1996.

[19] M. R. Palattella, N. Accettura, X. Vilajosana, T. Watteyne, L. A. Grieco, G. Boggia, and M. Dohler, "Standardized protocol stack for the internet of (important) things," *IEEE Communications Surveys Tutorials*, vol. 15, pp. 1389–1406, Third 2013.

[20] T. Graf, *Messaging over IPv6 destination options*, 2003. URL: `http://gray-world.net/papers/messip6.txt` (visited on 2020-05-23).

[21] N. B. Lucena, G. Lewandowski, and S. J. Chapin, "Covert Channels in IPv6," in *Privacy Enhancing Technologies* (G. Danezis and D. Martin, eds.), (Berlin, Heidelberg), pp. 147–166, Springer Berlin Heidelberg, 2006.

[22] D. Llamas, C. Allison, and A. Miller, "Covert channels in internet protocols: A survey," in *Proceedings of the 6th Annual Postgraduate Symposium about the Convergence of Telecommunications, Networking and Broadcasting, PGNET*, vol. 2005, 2005.

[23] W. Mazurczyk, S. Wendzel, S. Zander, A. Houmansadr, and K. Szczypiorski, *Information Hiding in Communication Networks: Fundamentals, Mechanisms, Applications, and Countermeasures*. Wiley-IEEE Press, 1st ed., 2016.

[24] K. S. Lee, H. Wang, and H. Weatherspoon, "PHY Covert Channels: Can you see the Idles?," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, (Seattle, WA), pp. 173–185, USENIX Association, Apr. 2014.

[25] Z. Wu, Z. Xu, and H. Wang, "Whispers in the Hyper-Space: High-Bandwidth and Reliable Covert Channel Attacks Inside the Cloud," *IEEE/ACM Transactions on Networking*, vol. 23, no. 2, pp. 603–615, 2015.

[26] O. Pearson, *DNS Tunnel - through bastion hosts*, 1998. URL: `https://www.gray-world.net/papers/dnstunnel.txt` (visited on 2020-05-25).

[27] E. Ekman, *iodine*, 2006. URL: `https://code.kryo.se/iodine/` (visited on 2020-05-25).

[28] W. Mazurczyk and L. Caviglione, "Information hiding as a challenge for malware detection," *IEEE Security Privacy*, vol. 13, no. 2, pp. 89–93, 2015.

[29] K. Cabaj, L. Caviglione, W. Mazurczyk, S. Wendzel, A. Woodward, and S. Zander, "The new threats of information hiding: The road ahead," *IT Professional*, vol. 20, no. 3, pp. 31–39, 2018.

[30] N. Tuptuk and S. Hailes, "Covert channel attacks in pervasive computing," in *2015 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pp. 236–242, 2015.

[31] L. Caviglione, A. Merlo, and M. Migliardi, "Covert channels in iot deployments through data hiding techniques," in *2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pp. 559–563, 2018.

[32] M. Guri, Y. Solewicz, A. Daidakulov, and Y. Elovici, "Acoustic data exfiltration from speakerless air-gapped computers via covert hard-drive noise ('diskfiltration')," in *Computer Security – ESORICS 2017* (S. N. Foley, D. Gollmann, and E. Snekkenes, eds.), (Cham), pp. 98–115, Springer International Publishing, 2017.

[33] A. Robles-Durazno, N. Moradpoor, J. McWhinnie, and G. Russell, "Waterleakage: A stealthy malware for data exfiltration on industrial control systems using visual channels*," in *2019 IEEE 15th International Conference on Control and Automation (ICCA)*, pp. 724–731, 2019.

[34] E. Carpentier, C. Thomasset, and J. Briffaut, "Bridging the gap: Data exfiltration in highly secured environments using bluetooth iots," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pp. 297–300, 2019.

[35] M. Guri, G. Kedma, A. Kachlon, and Y. Elovici, "Airhopper: Bridging the air-gap between isolated networks and mobile phones using radio frequencies," in *2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE)*, pp. 58–67, 2014.

[36] A. Nadler, A. Aminov, and A. Shabtai, "Detection of malicious and low throughput data exfiltration over the DNS protocol," *Computers & Security*, vol. 80, pp. 36 – 53, 2019.

[37] M. Ge, X. Fu, N. Syed, Z. Baig, G. Teo, and A. Robles-Kelly, "Deep learning-based intrusion detection for iot networks," in *2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pp. 256–25609, 2019.

[38] N. Koroniotis, N. Moustafa, E. Sitnikova, and B. Turnbull, "Towards the development of realistic botnet dataset in the internet of things for network forensic analytics: Bot-iot dataset," *Future Generation Computer Systems*, vol. 100, pp. 779 – 796, 2019.

[39] J. Postel, "Internet Control Message Protocol," RFC 792, Network Working Group, September 1981.

[40] D. Mills, J. Martin, J. Burbank, and W. Kasch, "Network time protocol version 4: Protocol and algorithms specification," RFC 5905, Internet Engineering Task Force (IETF), June 2010.

[41] T. Mizrahi and D. Mayer, "The Network Time Protocol Version 4 (NTPv4) Extension Fields," RFC 7822, Internet Engineering Task Force (IETF), March 2016.

[42] P. Mockapetris, "Domain Names - Implementation and Specification," RFC 1035, Network Working Group, November 1987.

[43] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose, "Resource records for the DNS security extensions," RFC 4034, Network Working Group, March 2005.

[44] A. Gulbrandsen, P. Vixie, and L. Esibov, "A DNS RR for specifying the location of services (DNS SRV)," RFC 2782, Network Working Group, February 2000.

[45] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," RFC 7252, Internet Engineering Task Force (IETF), June 2014.

[46] A. Banks, E. Briggs, K. Borgendale, and R. Gupta, "MQTT Version 5.0," Standard, OASIS Standard, March 2019.

[47] R. Godfrey, D. Ingham, and R. Schloming, "OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0," Standard, OASIS Standard, October 2012.

Appendix A

Hours invested

This work has been done during 5 months, tracking the time invested for every task. Figure A.1 shows a detailed view of the invested effort by week and task. This work consisted on (1) study of the state of the art regarding data exfiltration, (2) study of protocols, (3) protocol comparison, (4) Python library development, and (5) empirical protocol evaluation.

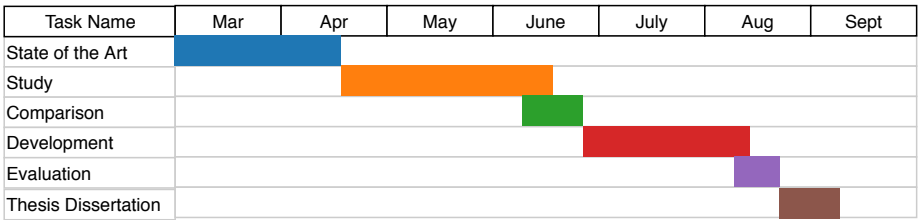


Figure A.1: Gantt chart with the effort invested on each task.

In summary, 634 hours have been invested in this work. Figure A.2 shows the effort dedicated to each task.

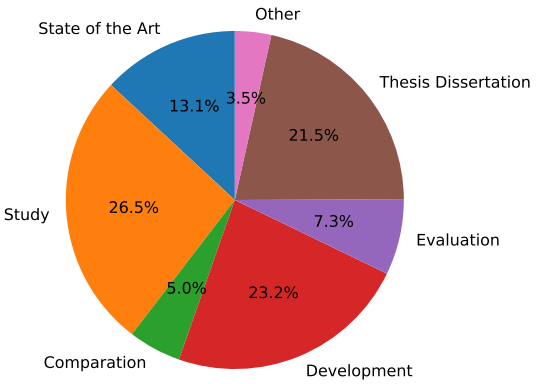


Figure A.2: Effort invested by task.