pySRUP – Simplifying Secure Communications for Command & Control in the Internet of Things

Andrew John Poulter*, Steven J. Johnston†, Simon J. Cox‡
University of Southampton,
Faculty of Engineering and the Environment,
Southampton, United Kingdom
Email: a.j.poulter@soton.ac.uk* sjj698@zepler.org† s.j.cox@soton.ac.uk‡

***Abstract*—Devices connected as a part of the Internet of Things, and other connected devices, require secure mechanisms to facilitate the propagation of Command & Control messages to enable the remote management of the devices. The Secure Remote Update Protocol (SRUP) provides a mechanism to facilitate the formation of secure messages for transmission using common techniques. This paper describes the details of, and motivations behind, the creation of a Python wrapper for SRUP: and describes a worked example of usage. By using pySRUP a developer can integrate SRUP into their projects: with minimal additional effort, and requiring them to write less code than traditional unsecured alternatives.**

***Index Terms*—MQTT; IoT; C2; Command & Control; Python; Security**

I. Introduction

The potential advantages of the Internet of Things (IoT) are constrained by the plethora of examples of connected devices which have been built using little or no security; or using out-dated protocols. For example, research published by cybersecurity company Kaspersky Lab in September 2018, showed that over 75% of attacks [1] attempted against their monitored honeypots, were attempting to exploit the fifty-year-old Telenet protocol.

The Secure Remote Update Protocol (SRUP) [2], [3], attempts to provide a solution to the problem of establishing secure and authenticated communications to IoT devices, by defining a protocol to enable Command and Control (C2) messages to be securely passed between IoT devices and C2 servers: using Message Queuing Telemetry Transport (MQTT) [4] as the application transport mechanism (and using Transport Layer Security (TLS) encryption to prevent eavesdropping). The original C++ library implementation of the protocol enables an IoT developer to focus on the device and application — and not need to design and implement their own security solution.

Although use of the SRUP protocol via the low-level implementaion addresses the security concerns: it remains as something that an applications developer is required to think about, and explicitly include within their system design; and despite the support from the library code, it remains somewhat non-trivial to implement.

This paper proposes a solution, through the use of a convenient code library; which enables a device developer to utilize secure communications — without needing to be exposed to the implementation details, and requires less code to be written than would be required for a naïve insecure solution.

II. Background to Work

A. The requirements for secure C2 for the IoT

Security in the context of C2 messages for the IoT is composed of three key elements: authenticity, integrity, and confidentiality.

1) Authenticity: Authenticity is the most important requirement for any C2 system. Before any device acts upon a command message from a controller it is essential that the device is able to verify that the command originates from a valid source. Similarly, if a data message is received by a controlling server it is necessary to be able to ascertain that the message has originated from a valid device (and to identify from which device it has been received).

2) Integrity: Techniques to validate the integrity of the message enable the receiver to ascertain not only that the message has originated from a valid sender; but also that the message itself has not been tampered with (or otherwise corrupted) during its propagation through the network. Establishing the integrity of a message also requires that attempts by an attacker to retransmit a previously sent (and therefore, originally valid) message (a so-called *replay attack*), can be detected and prevented.

3) Confidentiality: It is often required that C2 messages sent between controllers and devices, also be protected against eavesdropping by an attacker. Whilst this is not a universal requirement, it is commonly required; and protecting the confidentiality of a message by encryption also provides additional challenges to an attacker attempting to inject malicious data into the system.

B. Overview of SRUP

The SRUP protocol provides an efficient binary payload to be delivered to the device in question using MQTT. This MQTT message payload consists of a byte-string containing details of the SRUP message type, the parameters associated with the message, details of the sender; and a digital signature. Devices subscribe to MQTT topics associated with their identity, and use these to receive and publish messages pertaining to the device. A C2 server will subscribe to multiple topics: one for each of the devices that it controls; as well as a *server* *topic* — used by devices wishing to initiate communications with a new server.

The use of SRUP addresses the first two of the security requirements (authenticity and integrity) by the use of message signatures generated using asymmetric cryptographic algorithms, such as RSA [5]; and addresses the third through the use of TLS encrypted MQTT [6].

III. Motivation

A. Types of attack against the IoT

There are three major classes of attack against IoT devices: attacks to obtain the data from an IoT device; attacks to damage (or deny use of) the IoT device itself; and attacks against a device, in order to use it as an attack vector against another target.

1) Attacks on data: Attacks against the data within an IoT system take two forms: attacks to attempt to exfiltrate sensitive data from the system; and attempts to inject false data into the system.

A partial resolution to attacks against the data is provided by the use of encryption to protect the data in transit. By using suitable encryption algorithms, interception of the content of data flowing over a network can be prevented. Effective use of encryption also requires a systemic approach to security to protect the data. In addition to protecting the data in transit, it is necessary to prevent unauthorized users exploiting legitimate access mechanisms to obtain or generate data: such as by using authenticated access control. It is also required to protect against attempts to directly compromise the data-stores themselves (such as by limiting the available attack surface, and hardening interfaces).

Preventing an attacker from injecting false data into the system, can similarly be protected against by the use of asymmetric encryption; but again also requires mechanisms to protect against unauthorized access to systems.

Lastly it must be recognised that in the event that a device is covertly physically compromised, it is almost impossible to prevent an attacker from using it to obtain or inject data. If (or when) such an attack is detected, however, mechanisms can be used to limit further access — such as remotely deauthorizing the device, and preventing it from reconnecting with the system — and it may also be possible to audit stored data to remove records which have come from a compromised device.

2) Attacks against the device: Attacks against the device, with intent to damage or deny its use; or to damage deny use of other physical systems connected to the device are a growing threat for the IoT [7]. IoT devices (or connected hardware) may be physically damaged (such as was seen with the Stuxnet malware [8]), or may be rendered permanently unusable by corrupting their firmware [9]. This class of attack also includes *crypto-ramsonware* attacks such as WannaCry [10]; which although are not currently widespread, have the potential to be highly damaging [11].

Attacks against network infrastructure (such as denying use of Radio Frequency (RF) communications through the use of jamming or other types of electronic RF attack) may be used to deny use of a network-connected device; and physical attacks against the device my damage or destroy it. Mitigation against both of these threats require physical security guarantees for the operating environment — and as such, are beyond the scope of a software solution, and are therefore not considered further in this paper.

3) Attacks exploiting the device: Attacks against a device may not necessarily have the device itself as the intended end-point; but rather use the device as an attack vector to target another resource. This has been very commonly seen within the IoT — such as with the Mirai malware [12], used to control a botnet of IoT devices to coordinate a Distributed Denial of Service (DDoS) attack against the Internet's Domain Name System (DNS) in 2016.

B. Perception that security is hard

The proliferation of IoT devices on the market with little or no meaningful security associated with them [13], clearly shows the extent to which adequate security is not regarded as a high-priority for IoT device designers. Implementing a secure system from scratch not only requires a high degree of understanding of security principles and components; but also requires a significant additional effort over and above an insecure (or unsecured) configuration. As such, even if a device developer wants to create a secure device, there is a significant barrier to entry.

C. The rise of "high-productivity" programming

Software development today is increasingly about assembling extant components into assembled applications. This trend is exemplified by programming languages such as JavaScript and Python — where developers use *package management* tools to automatically retrieve and install libraries and their dependancies from which the application software is composed. This approach allows for far higher productivity from a software developer: allowing them to concentrate on the features required within the software, and not on the implementation of the lower-level capabilities required to enable them.

This progression, of increasing abstraction from the underlying hardware, is the continuation of a trend from the earliest days of computing. Today, almost no-one would consider developing serious application software by directly using machine instructions or writing assembly code mnemonics; and intermediate level languages (such as C or C++) are increasingly only used where speed of execution is critical. Python is often ranked as the 'top' programming language in surveys of real-world language use. [14].

IV. About pySRUP

A. Description of the pySRUP library

The pySRUP library provides a Python implementation of both the underlying SRUP protocol, and of the necessary MQTT and Secure Hyper-Text Transfer Protocol (HTTPS) clients required to handle the message receipt and delivery;

as well as the initial key exchange and device registration. Once the back-end services (such as the MQTT broker, and secure key exchange service) required to enable this are in place (see Section V); developers can consume the services they provide, without needing to implement their own device-side client software.

For example, a developer wishing to construct a device which can handle a SRUP *action* message — will simply write a function defining what (from the device's perspective) should occur on receipt of such a message: and provide this as a call-back function to the pySRUP client class object. The underlying library code will then handle the detailed implementation of the messaging process (such as receiving the underlying MQTT message, validating the message signature, and checking for an attempted replay attack). If the message passes the validation tests, then the details of the message will be passed handler function to perform the correct action.

For more complex message types (such as those used to perform a software update operation), the library can automatically handle the intermediate message exchanges: notifying the application software only when the retrieval and validation of the update data has been completed — and the *activation* signal has been received.

The library itself is structured in the form of two parts; firstly a simple Python wrapper for the extant C++ library implementation of SRUP, and on top of this, a pure Python implementation of the higher-level functionality (including interfacing with the Paho [15] MQTT client; and the Requests [16] HTTPS client). This abstracted library encapsulates the functionality in the form of two Python classes: providing distinct implementations for both devices, and controlling C2 server applications.

The advantage of this hierarchical approach is one of simplicity. It maximizes reuse of the extant C++ library code, and ensures that the best use is made of the high-productivity features of the Python language, and its ecosystem of pre-existing libraries.

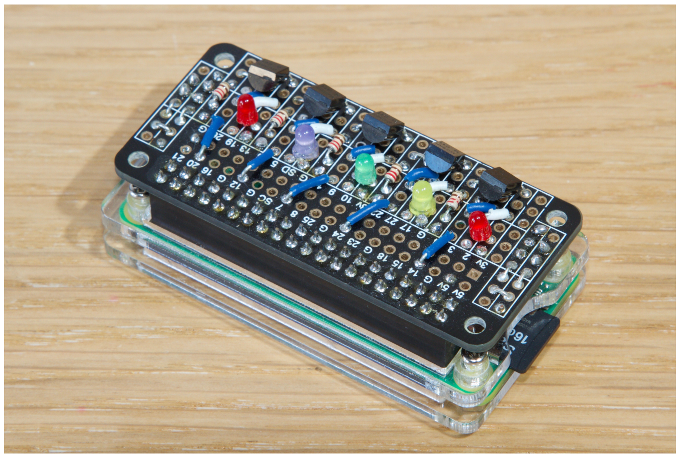


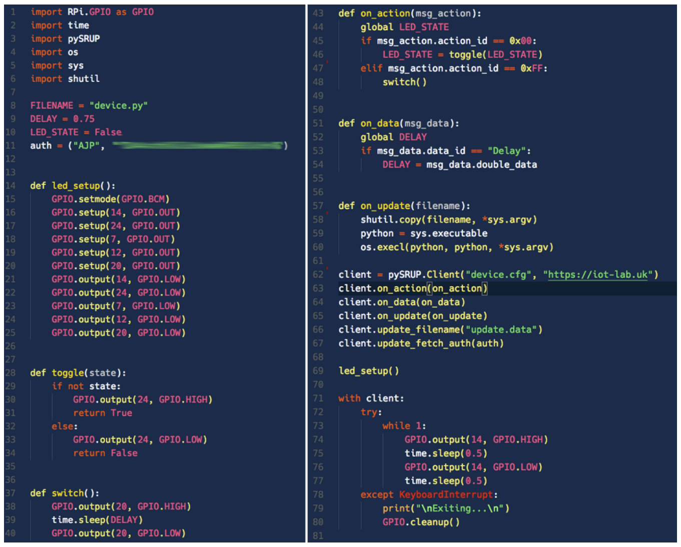
Fig. 1. A photograph of an example IoT device, consisting of a Raspberry Pi Zero W and a custom circuit board.

B. Ease of use comparison

Although the C++ library provides a full implementation of the SRUP protocol, in order to use it from C++, the developer must implement their own interface between it and an MQTT library. Similar the library does not provide any interface to the key exchange service (but rather assumes that the necessary key exchange has already taken place outside of the functionality of the library code). By contrast, when using the pySRUP library: such steps are handled automatically, so that a developer only needs to implement their own functionality.

In order to demonstrate the concept, an example IoT device, consisting of a Raspberry Pi Zero W — and a custom circuit-board with a number of Light Emitting Diodes (LEDs) — was constructed. This is shown in figure 1.

The device-side software for this device (which includes handling ACTION and DATA messages, as well as the ability to remotely update the software); was implemented in less than 100-lines of Python code, when using the pySRUP library (and over 20% of that code is preamble and setup for the General Purpose Input / Output (GPIO) to control the LEDs). Figure 2 illustrates the brevity of the code required to implement the device.


```python
import RPi.GPIO as GPIO
import time
import pySRUP
import os
import sys
import shutil

FILENAME = "device.py"
DELAY = 0.75
LED_STATE = False
auth = ("AJP", [redacted])


def led_setup():
    GPIO.setmode(GPIO.BCM)
    GPIO.setup(14, GPIO.OUT)
    GPIO.setup(24, GPIO.OUT)
    GPIO.setup(7, GPIO.OUT)
    GPIO.setup(12, GPIO.OUT)
    GPIO.setup(20, GPIO.OUT)
    GPIO.output(14, GPIO.LOW)
    GPIO.output(24, GPIO.LOW)
    GPIO.output(7, GPIO.LOW)
    GPIO.output(12, GPIO.LOW)
    GPIO.output(20, GPIO.LOW)


def toggle(state):
    if not state:
        GPIO.output(24, GPIO.HIGH)
        return True
    else:
        GPIO.output(24, GPIO.LOW)
        return False


def switch():
    GPIO.output(20, GPIO.HIGH)
    time.sleep(DELAY)
    GPIO.output(20, GPIO.LOW)
```

```python
def on_action(msg_action):
    global LED_STATE
    if msg_action.action_id == 0x00:
        LED_STATE = toggle(LED_STATE)
    elif msg_action.action_id == 0xFF:
        switch()


def on_data(msg_data):
    global DELAY
    if msg_data.data_id == "Delay":
        DELAY = msg_data.double_data


def on_update(filename):
    shutil.copy(filename, *sys.argv)
    python = sys.executable
    os.execl(python, python, *sys.argv)

client = pySRUP.Client("device.cfg", "https://iot-lab.uk")
client.on_action(on_action)
client.on_data(on_data)
client.on_update(on_update)
client.update_filename("update.data")
client.update_fetch_auth(auth)

led_setup()

with client:
    try:
        while 1:
            GPIO.output(14, GPIO.HIGH)
            time.sleep(0.5)
            GPIO.output(14, GPIO.LOW)
            time.sleep(0.5)
    except KeyboardInterrupt:
        print("\nExiting...\n")
        GPIO.cleanup()
```

Fig. 2. The device-side Python code for a simple IoT application, using the SRUP protocol to control a device: implemented using the pySRUP library.

When using a solution utilizing pySRUP, a developer is required to write less code than would be necessary to implement a simple unsecured communication model: because the pySRUP library takes care of all of the low-level setup of the MQTT client. A developer implementing even a simple MQTT-based communications model by directly using the same Paho MQTT library, would be required to explicitly connect to the broker and subscribe to suitable topics: and implement their own `on_message()` handler to parse the MQTT messages. When a message is received, this would be required identify the correct behaviour to perform. Both of these elements require considerably more code and more consideration than an approach based on pySRUP.

C. *Why Python?*

Given the prevalence of Python in the current software trends, its ease of use; and the ease to which Python can interface with C/C++ code — it is a very good fit for a system such as described here.

By interfacing with the existing C++ library, rather than reimplementing it in Python, it is possible to both reuse the code — and also to allow the lower-level work (such as marshalling & demarshalling bytes, and calculating & checking cryptographic signatures) to be performed in a language optimized for speed and efficient byte-level operations on data.

Extant MQTT and Hyper-Text Transfer Protocol (HTTP) libraries (such as Paho and Requests) make it extremely easy to interface with the backend systems; and utilizing cryptographic libraries such as PyCryptodome [17] make non time-critical processes, such as key generation, simple to implement.

V. A RAPIDLY DEPLOYABLE, CONTAINERIZED, MICRO-SERVICES ARCHITECTURE

A. *Back-end Architecture*

The back-end architectural requirements for a system using SRUP are straightforward. The requirement is only for an MQTT broker; and an HTTPS web application to facilitate the key exchange. In practice it is expected that in most applications the C2 service would also be delivered as a part of the back-end web services.

There needs to be close coupling between the MQTT broker and the key exchange service if TLS is being used to encrypt the MQTT traffic. It is necessary, not only to ensure that the key exchange service is also issuing access keys to be used when establishing a secure connection to the broker; but also to ensure that the broker is configured only to permit devices to access their own topics. Without this constraint, the security of the configuration could be easily by-passed by an attacker, registering via the key exchange service (to obtain an access key) — and then using it to subscribe to the topics associated with other devices. Most MQTT brokers support the concept of an *Access Control List (ACL)* — which constrains which topics a device may subscribe and / or publish to.

The MQTT broker can be configured to only permit MQTT clients presenting a valid X.509 [18] certificate, which has been issued by a Certificate Authority (CA) with a common trust to the CA used to sign the certificate used by the broker.

To further restrict access on a per-topic basis, the certificate can be issued with the Common Name (CN) field set to equal the device's unique identifier (a 128-bit Universally Unique Identifier (UUID) value assigned to the device, as a part of the registration & key exchange phase). The broker can then use this CN data as the broker access username; and the ACL can then be easily configured to restrict read access to topics, on the basis of this username. The ACL uses a wildcard system: such that all devices connecting with valid certificates, may only access their own individual topics.

In order to permit devices to contact a new C2 server, for the purposes of *joining* a new C2 node — devices must also be granted write-only access to topics associated with servers.

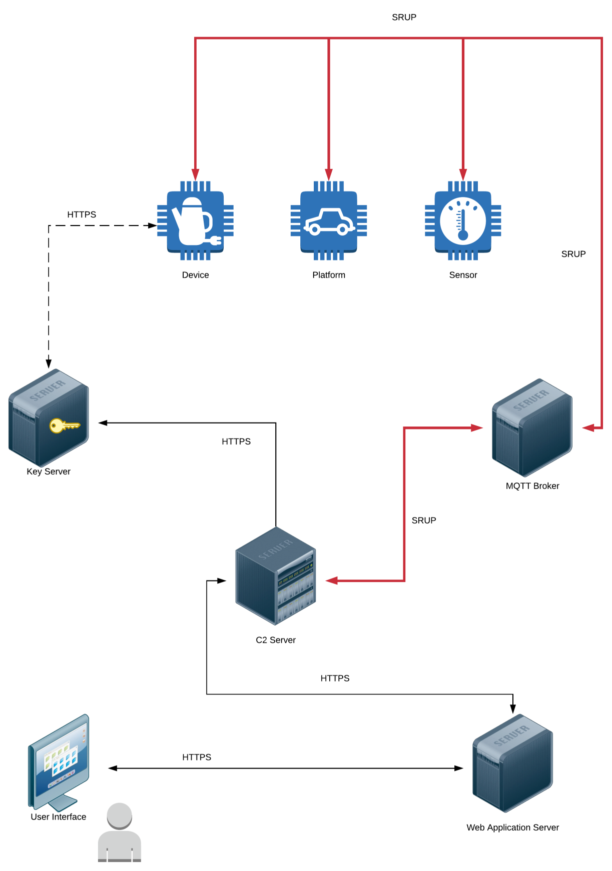


Fig. 3. A diagram showing the required back-end architecture for a system using the SRUP protocol

Although the MQTT standard does not support error reporting when a device attempts to subscribe to a disallowed topic: the broker's ACL will prevent communications from being received by received by an unauthorized device.

B. *Towards containerization*

Although the current version of the SRUP code contains the pySRUP library, and an implementation of the key exchange service: an end-user is required to construct their own back-end services. It is proposed to provide, in the future, a reference implementation of the back-end in the form of a series of Docker containers. Docker [19] is the most widespread containerization system in use today, and using such a containerized approach eliminates the need for an end-user to need to deploy and correctly configure their own broker and other services. Using a model where the pre-configured broker, and the key exchange service are implemented as discrete containers is consistent with a modern trend for cloud-native, micro-service based architectures. For ease of management these containers (together with other additional functionality which may be required: such as a web-based C2 web application, or a HTTPS web server to be used as a part of the software update process) may be orchestrated for automated deployment using a Kubernetes pod [20].

VI. Conclusions

This paper introduced the pySRUP library, and has shown how this can be used in order to facilitate rapid development of Python-based IoT devices. We have shown how efficiently an implementation of a simple, highly secure, example application may be created using pySRUP; and we have shown that the quantity of code that a developer is required to write in order to utilize SRUP is equivalent to (or even less-than) the code required to implement an unsecured solution.

The work to-date has all been implemented using Raspberry Pi based systems: but the code should be compatible with any CPython based implementations. Further work is underway to assess the feasibility of porting SRUP and the pySRUP library to the MicroPython framework — in order to facilitate its use on lower-level, microcontroller-based, devices.

In order to encourage use of the SRUP protocol, all of the code associated with this research has been released as open source source, released under the terms of the MIT Licence.

The full version of the application code shown in figure 2 can be obtained from:

DOI: 10.5281/zenodo.2575652

The most recent version of the full library code may be obtained from:

https://github.com/dstl/srup

DOI: 10.5281/zenodo.2575622

Acknowledgments

The lead author is a Principal Computer Scientist with the UK Ministry of Defence Science & Technology Laboratory (Dstl) who have funded this research. The authors would like to thank Dstl for their co-operation & support with this research.

References

[1] M. Kuzin, Y. Shmelev, and V. Kuskov. (2018, Sep.) New trends in the world of IoT threats. [Online]. Available: https://securelist.com/new-trends-in-the-world-of-iot-threats/87991/

[2] A. J. Poulter, S. J. Johnston, and S. J. Cox, "SRUP: The secure remote update protocol," in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*. IEEE, 2016, pp. 42–47. [Online]. Available: http://ieeexplore.ieee.org/document/7845397/

[3] A. J. Poulter, S. J. Johnson, and S. J. Cox, "Extensions and Enhancements to the Secure Remote Update Protocol," *Future Internet*, vol. 9, no. 4, p. 59, Sep. 2017. [Online]. Available: http://www.mdpi.com/1999-5903/9/4/59/htm

[4] A. Banks and R. Gupta, "MQTT Version 3.1.1," Oct. 2014. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

[5] D. Boneh, "Twenty Years of Attacks on the RSA Cryptosystem," *Notices of the American Mathematical Society*, vol. 46, no. 2, pp. 203–213, Feb. 1999. [Online]. Available: http://www.ams.org/notices/199902/boneh.pdf

[6] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," Internet Engineering Task Force, Mar. 2018. [Online]. Available: https://tools.ietf.org/html/draft-ietf-tls-tls13-28

[7] C. Edwards, "IoT security threat as embedded systems struggle," *IET Engineering and Technology Magazine*, Feb. 2016. [Online]. Available: https://eandt.theiet.org/content/articles/2016/02/iot-security-threat-as-embedded-systems-struggle-1/

[8] D. Kushner, "The real story of stuxnet," *IEEE Spectrum*, pp. 48–53, Mar. 2013. [Online]. Available: http://ieeexplore.ieee.org/document/6471059/

[9] Radware, "BrickerBot PDoS Attacks: Back With a Vengeance," Tech. Rep., Apr. 2017. [Online]. Available: https://security.radware.com/ddos-threats-attacks/brickerbot-pdos-back-with-vengeance/

[10] Q. Chen and R. A. Bridges, "Automated Behavioral Analysis of Malware: A Case Study of WannaCry Ransomware," in *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA) IS - SN - VO - VL -*. IEEE, Dec. 2017, pp. 454–460. [Online]. Available: http://ieeexplore.ieee.org/document/8260673/

[11] C. Wüest, "Is Ransomware coming to IoT devices?" in *CRESTcon & IISP Congress Conference 2016*, Mar. 2016. [Online]. Available: http://www.crestandiisp.com/wp-content/uploads/2016/03/CandidWueest.pdf

[12] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "DDoS in the IoT: Mirai and Other Botnets," *IEEE Computer*, vol. 50, no. 7, pp. 80–84, 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7971869/

[13] N. Dragoni, A. Giaretta, and M. Mazzara, "The Internet of Hackable Things," in *Proceedings of 5th International Conference in Software Engineering for Defence Applications*, P. Ciancarini, S. Litvinov, A. Messina, A. Sillitti, and G. Succi, Eds. Cham: Springer International Publishing, 2018, pp. 129–140. [Online]. Available: https://arxiv.org/pdf/1707.08380.pdf

[14] S. Cass, "The 2018 Top Programming Languages ," *IEEE Spectrum*, Jul. 2018. [Online]. Available: https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages

[15] Eclipse. Eclipse Paho - MQTT and MQTT-SN software. [Online]. Available: https://www.eclipse.org/paho/clients/python/

[16] K. Reitz. (2015) Requests: HTTP for Humans — Requests 2.9.1 documentation. [Online]. Available: http://docs.python-requests.org/en/latest/

[17] H. Eijs. PyCryptodome. [Online]. Available: https://github.com/Legrandin/pycryptodome

[18] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, W. Polk, and R. Housley, "Internet X. 509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile." Internet Engineering Task Force, May 2008. [Online]. Available: https://tools.ietf.org/pdf/rfc5280.pdf

[19] Docker. (2015, May) What is Docker? [Online]. Available: https://www.docker.com/what-docker

[20] The Kubernetes Authors. (2018, Sep.) What is Kubernetes? [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/