UNIVERSITY OF SOUTHAMPTON

Faculty of Engineering & Physical Sciences
School of Engineering

Engineering Applications and Architectures for Cybersecurity of Command-and-Control Messaging in the Internet of Things.

by

Andrew John Poulter
CEng. MSc. BSc. (Hons)
ORCiD: 0000-0002-3438-3981

A thesis for the degree of
Doctor of Philosophy

February 2022

University of Southampton

Abstract

Faculty of Engineering & Physical Sciences
School of Engineering

Doctor of Philosophy

Engineering Applications and Architectures for Cybersecurity of Command-and-Control Messaging in the Internet of Things.

by Andrew John Poulter

This thesis explores ideas connected with the cybersecurity of, and secure communications for, Internet of Things (IoT) devices; and introduces a number of original elements of research — including the Secure Remote Update Protocol (SRUP), a protocol developed to provide a mechanism for secure Command and Control messages.

The work introduces cybersecurity concepts and background, IoT networking protocols and Command and Control messaging, before moving on to describe the original research.

The design and concept of SRUP is described in detail, along with a scheme to support the use of dynamic identity in the context of the IoT. Techniques to establish device identity are then described, followed by an examination of the security features of SRUP.

An open-source implementation of SRUP is then introduced, alongside a discussion on the way this has been optimized for ease of use by non-specialist developers. A concept to enable the controlled sharing of information and requests between Command and Control networks using SRUP is then described, along with a discussion on how this approach could be adopted to help to address the problem of short-term provision of access to IoT systems by guest users.

Finally an experimental assessment of the protocol in simulated real-world conditions is described and measurements of the performance overhead associated with using SRUP, with inexpensive low-power hardware, are discussed and analysed. These results show that the use of the SRUP protocol, in comparison to an insecure implementation, added an additional processing delay of between 42.92ms and 51.60ms to the end-to-end message propagation — depending on the specific hardware in use.

The thesis concludes with a summary of the research, and some recommendations for follow-on work.

Contents

List of Figures

List of Tables

Listings

Declaration of Authorship

I declare that this thesis and the work presented in it is my own and has been generated by me as the result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published as:
    - A. J. Poulter, S. J. Johnston, and S. J. Cox, “Using the MEAN Stack to Implement a RESTful Service for an Internet of Things Application,” in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, Jan. 2015, pp. 280–285. DOI: 10.1109/WF-IoT.2015.7389066
    - 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, Jan. 2016, pp. 42–47. DOI: 10.1109/WF-IoT.2016.7845397
    - A. J. Poulter, S. J. Johnston, and S. J. Cox, “Extensions and Enhancements to The Secure Remote Update Protocol,” *Future Internet*, vol. 9, no. 4, p. 59, Sep. 2017. DOI: 10.3390/fi9040059
    - A. J. Poulter, S. J. Johnston, and S. J. Cox, “pySRUP – Simplifying Secure Communications for Command Control in the Internet of Things,” in *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*, Apr. 2019, pp. 273–277. DOI: 10.1109/WF-IoT.2019.8767205

- A. J. Poulter, S. Johnston, and S. Cox, “Secure Messaging, Key Management & Device identity for the IoT,” in *Presentation to IoT Security Foundation Conference 2019*, IoT Security Foundation, Nov. 2019. [Online]. Available: https://youtu.be/vdjY617WvHo
- A. J. Poulter, S. J. Ossont, and S. J. Cox, “Enabling the Secure Use of Dynamic Identity for the Internet of Things—Using the Secure Remote Update Protocol (SRUP),” *Future Internet*, vol. 12, no. 8, p. 138, Aug. 2020. DOI: 10.3390/fi12080138
- A. J. Poulter and S. J. Cox, “Enabling secure guest access for Command-and-Control of Internet of Things devices,” *IoT*, vol. 2, no. 2, pp. 236–248, 2021. DOI: 10.3390/iot2020013
- A. J. Poulter and S. J. Cox, “An assessment of the performance of the secure remote update protocol in simulated real-world conditions,” *IoT*, vol. 2, no. 4, pp. 549–563, 2021. DOI: 10.3390/iot2040028

Signed:.. Date:9th February 2022

Acknowledgements

This work has been funded by the Defence Science and Technology Laboratory (Dstl). Dstl is a part of the United Kingdom Ministry of Defence.

The author is employed by Dstl as a Principal Engineer.

To my wife. . .

Part I

Background

Chapter 1

Introduction

This introductory Chapter outlines the research questions that this work is seeking to address, the background requirements and context for this work, and describes the research methodology that has been adopted.

1.1 Outline, research questions, and thesis structure

Since 2010 the total number of Internet of Things (IoT) and other connected devices has grown by more than 10% per year, from around 800 million device connections in 2010 to over 10 billion in 2020 [9]. IoT devices are becoming increasingly ubiquitous in the modern world, and as such the need for a solution to provide secure mechanisms to enable the Command and Control (C2) of the ensuing complex systems is ever growing.

This research proposes methods to facilitate secure C2 operations for the IoT through the design, implementation, and demonstration of an open-source protocol and supporting architectures to enable secure messaging.

1.1.1 Research questions

Specifically, the research questions that this work seeks to answer are:

RQ1 How can a secure protocol for Command and Control messaging for the Internet of Things be developed from extant, tried and tested, commodity software and network communications components?

RQ2 How can such a protocol be used to enable automated secure key distribution and identity management?

RQ3 How can such a protocol (and associated architectures) be used to provide assurance around the identity of a physical device?

RQ4 Can the protocol be made robust to attempted attacks against it, and its supporting infrastructure?

RQ5 How can the protocol be made sufficiently easy to use, that it becomes simpler for a prospective user to adopt the secure protocol, than to implement an insecure system?

RQ6 Can such a protocol be extended to provide a mechanism to securely share controlled subsets of data between Command and Control systems, whilst enabling system owners to retain overall control?

RQ7 What is the performance overhead of such a protocol when compared to insecure methods, and how well does it cope with poor network conditions?

1.1.2 Structure of this thesis

The remainder of this chapter outlines the background to the research questions and introduces the concepts of connected digital devices, the Internet of Things, and discusses their context and background providing justification as to the requirement for this work.

Subsequent chapters of this thesis:

- Introduce the topic of cybersecurity, describe a number of commonly exploited vulnerabilities and types of malware, and explore the need for specific protections for the IoT (Chapter 2)
- Describe in detail the communications protocols and architectures that are commonly used for Internet and IoT applications today (Chapter 3)
- Describe the concept of C2, and the requirements for C2 messaging for the IoT (Chapter 4)
- Propose a novel application-layer protocol for secure and authenticated IoT communications (Chapter 5) [RQ1]
- Explore the specific issues with regards to the key distribution and identity management within a C2 network (Chapter 6) [RQ2]
- Describe C2 network operations, and the use of a *moderated* network joining process to provide assurance of device identity (Chapter 7) [RQ3]
- Examine the security design of the protocol introduced in Chapter 5, and discuss how the associated threats can be mitigated (Chapter 8) [RQ4]

- Describe in detail the approach taken to design and build a software implementation of the protocol and the necessary supporting infrastructure, in order to facilitate ease-of-use (Chapter 9) [RQ5]
- Explore how the protocol can be extended to include features to enable sharing of data and control (Chapter 10) [RQ6]
- Assess the performance of the protocol in simulated real-world conditions, and perform comparisons to other methods of IoT device connectivity (Chapter 11) [RQ7]

Finally, Chapter 12 draws conclusions about this work and makes recommendations for follow-on research work.

1.1.3 Research methodology

This research has adopted an iterative, experimental approach to solving the research questions. Having developed a theoretical protocol, and hypothesized to address the requirements, the approach taken has been to then evaluate the concepts experimentally. A number of iterations of the software have been developed, culminating in the feature-complete version described in this thesis.

Experimental work has been conducted to subject increasingly feature-complete versions of the software implementation of the protocol, to increasingly formalized evaluation, in increasingly realistic use-case scenarios; the conceptual model of the protocol being modified subject to the outcome of the experimental assessment.

1.1.4 The effects of the COVID-19 pandemic on this work

Although the ongoing global COVID-19 (SARS-CoV-2) pandemic has caused only limited disruption to the originally proposed schedule of work, it has curtailed the degree to which it was possible to implement a fully field-deployed final experiment. As such, performance assessments of the protocol were conducted in a laboratory setting, as was the *capstone* demonstration of all aspects of the protocol in action — which ultimately consisted of a bench-top deployment of the software and associated custom hardware, in a laboratory setting.

1.2 Background and concepts

In order to understand the requirements for a solution to address the research questions, there are two main areas of background:

- The concept of *commodity components* and their use within the Internet of Things
- A conceptual understanding of *connected digital devices*

1.2.1 Commodity components

The term commodity components refers to mass-market and widely available and used hardware or software, and this is often considered in the context of High Performance Computing (HPC). Since the inception of the *Beowulf cluster* in 1995 [10], and early work on Windows HPC [11], the use of commodity hardware to build large-scale HPC clusters has become dominant [12].

The use of commodity hardware has also been recently seen at the other end of the compute spectrum, with the development of the Raspberry Pi single board computer [13], originally developed to produce an affordable computer that young people could use to learn computer programming [14]. Due to its low-cost and small size, the Raspberry Pi has become near-ubiquitous as a single-board computer within experimental and home-made IoT devices in fields as diverse as home-automation, environmental monitoring, and robotics [15]–[17]. Work has even shown that viable (demonstration) HPC clusters can be built using Raspberry Pi hardware [18].

Even traditionally conservative disciplines such as aerospace have recently started to adopt the use of commodity hardware and software. In June 2020, engineers from SpaceX answered questions on the online discussion forum *Reddit*, [1] describing how they make use of Open Source Software (OSS) in the form of the Linux Operating System, and run applications developed in C++, Python, and even JavaScript for different aspects of the systems, automation and user-interface on their *Crew Dragon* spacecraft [19]. Such approaches are not unique to disruptive non-traditional vendors. Even the United States' National Aeronautics and Space Administration (NASA) made use of a commodity Central Processing Unit (CPU), commodity sensor hardware, and Linux as a part of their experimental *Ingenuity* Mars rover helicopter [20].

The same approach can be considered when looking at communications protocols such as Hyper-Text Transfer Protocol (HTTP) [21], OpenSSL [22], Transport Layer Security (TLS) [23] and Message Queuing Telemetry Transport (MQTT) [24], which are widely exploited within the IoT [25], [26].

This research seeks to utilize extant and commonly-used software, libraries, and protocols such as these as the basis for a novel set of methods to enable secure communications to and from IoT devices.

[1] https://www.reddit.com/r/spacex/comments/gxb7j1/we_are_the_spacex_software_team_ask_us_anything/

1.2.2 Connected digital devices

The decreasing size and cost, and increasing power of microprocessors over the last fifteen-years has led to the very widespread adoption of microelectronic systems, approaching or attaining the *ubiquitous computing* concept proposed by Weiser [27] and further developed at the Xerox Palo Alto Research Center (PARC) [28]. Co-incident with the fall in the cost of microprocessors, this time period has also seen the increased ubiquity of Internet Protocol (IP) network connections.

Although other communications standards and systems are available, the use of IP networks is so ubiquitous that even systems using other networking protocols almost invariably make use of a gateway, to link back to an IP network to permit greater interoperability and interaction with the myriad of other Internet connected digital devices. As such, although some architectures for connected digital devices use other networking standards to link to and from the physical device, they nearly all adopt an IP network for onward connection. IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) [29] is perhaps the best known example of this with nodes addressable via Internet Protocol Version 6 (IPv6) [30], but exploiting meshed low-power, low-rate, *Personal Area Networks* using the IEEE 802.15.4 standard [31] on the devices themselves.

The fusion of the availability of networking and the power of low-cost microprocessors has created a world of special-purpose connected digital devices. The idea of connecting something other than a general purpose computing device to the Internet is not fundamentally new. For example, computer network connected vending machines pre-date both the world wide web, and even much of the Internet itself, being first seen experimentally in the 1970s [32]; but in recent years the low-cost of microelectronics, and the widespread adoption of wireless networking such as WiFi [33], has led to these devices becoming widespread and genuinely useful.

1.2.3 The Internet of Things

As with many technical phrases which become adopted by the popular media, there is no clear single technical definition of the Internet of Things. In its earliest widespread usage [34], [35] it referred specifically to machine identification of objects by use of Radio-Frequency Identification (RFID) tags, whereas today it typically refers to a much broader set of network connected devices [36]. The use of Machine-to-Machine (M2M) communications, using standardized commodity communications technologies has been growing since the early part of this millennium [37].

The IoT also has ancestry in the concept of *ubiquitous computing* first proposed by Weiser in the early 1990s [27], although the IoT is different to the original vision of ubiquitous computing — there is much similarity in the prevalence and *invisibility* of connected devices.

The phrase IoT as it is used today, can be taken to describe both consumer electronics devices, typically within a domestic or commercial setting; as well as networked industrial control systems — also known as the Industrial Internet of Things (IIoT) [38].

Perhaps one of the most effective definitions of the IoT in the form we see it today is from security researcher Bruce Schneier, who described the IoT as *"...the network of physical objects that contain embedded technology to communicate and sense or interact with their internal states or the external environment"*, in his 2018 book '*Click Here to Kill Everyone*' [39].

There are a range of estimates of the number of deployed IoT devices in operation today, and many predictions as to the future scale of the IoT. One of the most widely cited numbers is from 2016, when technology research firm Juniper Research, predicted over 46 Billion connected devices by 2021 [40], and in 2020 they concluded that there were already 35.7 billion devices in 2019, and predicted 83 billion by 2024 [41].

The availability of IP networks is today more prevalent than ever. The widespread deployment of Third-Generation (3G) and Fourth-Generation (4G) broadband cellular data services, and the growing deployment of Fifth-Generation (5G) systems today, make it possible to establish high bit rate IP connections from increasingly many locations around the world. There has also been a widespread proliferation of the provision of public WiFi technology in urban areas in recent years [42]. These, combined with the advent of low-cost satellite-based Internet provision such as SpaceX's *Starlink* [43], mean that there will soon be very few locations on Earth where it is not possible to obtain a connection to the Internet for a relatively low-cost. Directly connected IoT devices may be deployed without requiring any additional infrastructure, and global satellite-based Internet coverage will facilitate IoT connectivity anywhere on the globe [44] [45].

Consequentially today it is no longer unusual for sensors and devices (even in remote locations) to have access to the Internet [46]. The advent of IPv6 means that these devices can be directly addressable, rather than relying on Network Address Translation (NAT) [47].

Despite the growth in high-bandwidth connectivity, one of the primary requirements for any IoT device (especially a device that is designed to be operated in a remote location) is that it makes efficient use of the potentially constrained network bandwidth, and is robust to periods of poor connection or network outages [48].

For the purposes of this thesis, the term IoT communications shall be defined as pertaining specifically to **Machine-to-Machine communications between both traditional, and non-traditional computing devices, taking place over IP networks**.

Perhaps the archetypal example of an IoT appliance today is a smart lightbulb [49]. Such devices are commonplace and inexpensive. They typically enable a third-party, general-purpose computing device (often a smartphone, tablet, or smart watch) to provide the Human-Computer Interface (HCI) to the device: so that a human operator may use a more convenient HCI than may be provided directly by the device itself, or the infrastructure within which it is operating. However, this is in reality a poor example of the true benefits of the IoT, since it does not fully demonstrate the integration and the added-value of M2M communications beyond that offered by simply providing a remote interface to the device.

A much better example of an IoT device is an Internet-connected smart thermostat. Many of these devices combine both an external HCI, as well as true M2M communication. For example, a smart thermostat can change its behaviour based on the combination of its own sensors, additional external (potentially 3rd-party) sensors, the information it receives from a human operator, and data from a weather forecast service. Whilst this type of rich M2M communication is not essentially new, the difference is that today information is crossing the boundary of a single device and being supplied to *ad hoc* systems formed from third-party devices never originally designed to work together; via openly accessible services and interfaces, built using open communications standards.

1.2.4 Military applications of the Internet of Things

The United States Army Research Laboratory (US ARL) have described the concept of the Internet of Battle Things (IoBT) [50]. In this concept they propose that future network-centric warfare operations [51] will consist of large-scale augmentation of a future battlefield environment by IoBT devices, including: *"...sensors, munitions, weapons, vehicles, robots, and human-wearable devices"* [50]. IoBT will be deployed to directly create military effect; as well as to provide logistical support information to both human war-fighters, and other devices. It is likely that physical IoBT devices will be deployed across all three traditional war-fighting domains (maritime, land & air) [52], as well as the newly considered space domain [53], and that logical components of the IoBT will also be deployed within the cyberspace domain [54].

Not all military applications of IoT fall into this category of IoBT however. There is considerable military interest in the use of IoT technology in non war-fighting areas such as training, logistics and facilities management [55]. It is likely that many of these non war-fighting applications will exploit extant civilian IoT systems, rather than bespoke military systems. More conventional IoT devices may also be used in a military context in lower-intensity scenarios such as stability or peace keeping operations, or for specialist applications such as wide-area sensing. In some scenarios military and civilian IoT systems may be used alongside each other — for example in Military Aid to the Civil Authorities (MACA) operations such as disaster-relief [56].

A key difference between military applications of IoT technology, and those of the civilian world is that military IoBT devices in a war-fighting application will be subject to the hostile actions of an adversary, who can be assumed to be prepared to engage in any type of action to achieve effect. These include overt physical and kinetic effects to disrupt an IoBT network in addition to the more covert types of cyber operations expected when considering the security of other military, or civilian IoT devices.

1.2.5 Cyber Physical Systems

The term Cyber Physical Systems was first used around 2009 [57], and is defined by Wang, Ye, Xu, *et al.* as consisting of *"... two major components, a physical process and a cyber system. Typically, the physical process is monitored or controlled by the cyber system, which is a networked system of several tiny devices with sensing, computing and communication (often wireless) capabilities. The physical process involved may be a natural phenomenon (e.g. a dormant volcano), a man-made physical system (e.g. a surgical room) or a more complex combination of the two."* [58]

Although contemporarily the Cyber Physical Systems (CPS) themselves are very often networked to wider information systems, CPS are not necessarily required to be connected to external systems.

CPS is a broad category covering both industrial and domestic applications, the term CPS is most typically associated with devices used in industrial control applications, or devices used to control other physical infrastructure [59].

1.2.6 The defence and national security challenges of the IoT and Cyber Physical Systems

In December 2014, HM Government's Chief Scientific Advisor, Sir Mark Walport wrote: *"The Internet of Things has the potential to have a greater impact on society than the first digital revolution. There are more connected objects than people on the planet. The networks and data that flow from them will support an extraordinary range of applications and economic opportunities. However, as with any new technology, there is the potential for significant challenges too. In the case of the Internet of Things, breaches of security and privacy have the greatest potential for causing harm. It is crucial that the scientists, programmers and entrepreneurs who are leading the research, development and creation of the new businesses implement the technology responsibly. Equally, policy makers can support responsible innovation and decide whether and how to legislate or regulate as necessary. Everyone involved in the Internet of Things should be constantly scanning the horizon to anticipate and prevent, rather than deal with unforeseen consequences in retrospect."* [60].

The report goes on to make clear that one of the major challenges of the IoT is security — and that this must be designed into any IoT systems from the outset: *“. . . data governance and security considerations are not optional extras but should be considered at the beginning, and throughout the lifecycle of Internet of Things applications . . . ”* [60].

There is a particular concern when using any IoT device in the context of a military or security application. Clearly any IoBT or other IoT device being used operationally would need extremely high standards of security and encryption to protect what would be very highly sensitive information, but even in a non-operational context the use of IoT devices and systems need to be very carefully protected against any form of compromise by malicious individuals, politically motivated organizations, or hostile nation-states.

1.2.7 Critical National Infrastructure

Within the United Kingdom, HM Government defines Critical National Infrastructure (CNI) as: *“Those critical elements of national infrastructure (facilities, systems, sites, property, information, people, networks and processes), the loss or compromise of which would result in major detrimental impact on the availability, delivery or integrity of essential services, leading to severe economic or social consequences or to loss of life.”* [61]

In the modern connected world, even devices which form part of CNI may expect, or be desired, to be connected to the Internet. Thus, in the context of national security, the CPS and IoT elements of CNI require a very high degree of protection. Although there are no specific additional requirements for the security for CNI applications, the criticality of the systems involved do require that an extremely high standard of security is applied.

There can, however, be nationally significant infrastructure run by private companies, and attacks against these may have societal implications far beyond those to the operating company or its direct customers. An example of this was the May 2021 cyber attack on the Colonial Pipeline Company, which disrupted both commercial aviation and consumer petrol supplies across the south-eastern United States for six-days, and to remedy which the company reportedly paid a ransom of $4.4M [62].

1.3 Summary

This section has described the research questions this work is seeking to address, and described the research methodology. It has also introduced a number of concepts relating to the Internet of Things and Cyber Physical Systems. These concepts establish the context for this work. The next Chapter will explore the topic of cybersecurity in general, and some specific issues pertaining to cybersecurity within the IoT.

Chapter 2

Cybersecurity and The Internet of Things

This Chapter will discuss cybersecurity, examining a short history of cybersecurity threats and malicious software. It will also explore some specific ways that these cybersecurity threats and concepts can apply to the IoT; as well as discussing encryption as it pertains to modern IoT devices. This Chapter establishes the context for the research, and frames the requirement for secure IoT communications.

2.1 Cybersecurity

Although seldom formally defined cybersecurity [63] or Information Security pertain to the threats to the integrity of computer systems, the information they contain, and their hardware and other equipment connected to them. Although often used interchangeably some, such as von Solms and van Niekerk, argue that the two terms are different to each other [64]. More generally the phrase refers to resilience to types of attack against computer systems — either by malicious software (known as *malware*) or by direct attack from a malicious individual or group.

2.1.1 Cybersecurity as an element of national security

In a Government, military, and national security context, the concept of computers requiring protection from espionage or other hostile action dates back to the mid 1960s, but the more modern Defence and Security context to Cybersecurity has its roots in a 1997 United States' Department of Defense (DoD) wargame, *Exercise ELIGIBLE RECEIVER* [65].

Addressing the United States Senate Governmental Affairs Committee 'Hearing on Vulnerabilities of the National Information Infrastructure' in June 1998, Lieutenant General

Kenneth A. Minihan, the then Director of the United States National Security Agency (NSA) said that: *"As Exercise ELIGIBLE RECEIVER 97 graphically demonstrated, a moderately sophisticated adversary can cause considerable damage with fewer than thirty people and a nominal amount of money if the systems they are attacking are not adequately protected and defended."* [66].

A real-world penetration of United States Air Force (USAF) and DoD computer systems took place in the following year. Code-named *SOLAR SUNRISE* by the DoD, the attack (by a group of teenagers) highlighted the reality of the threat shown by ELIGIBLE RECEIVER [67]. ELIGIBLE RECEIVER, and SOLAR SUNRISE did much to raise public awareness of cybersecurity, especially in the United States [68].

2.2 Malware

The word malware is a simple portmanteau of the words malicious software, and refers to any software that is written with malicious intent [69]. Although malware itself is an ever growing phenomena, the specific threat types (especially the threat from the early types of malware such as viruses) have changed over time, with today's malware having *"...become the primary medium to launch large-scale attacks..."* [70].

2.2.1 Viruses

Although popularly used interchangeably with malware, the term computer viruses specifically refers to a special case of malware that is *self-replicating*. The first examples of Personal Computer (PC) malware were viruses [69], designed to replicate via the boot sector of floppy disks. The `Brain.A` virus was the first PC virus, and was released in 1986. It targeted Microsoft Disk Operating System (DOS) PCs, although it was more of a proof-of-concept since it had no malicious payload associated with it. By 1998 the United States Department of Energy (DoE) had identified and described more than 700 distinct viruses for DOS, and had started to document viruses targeting Microsoft's new Operating System (OS), Windows 95 [71]. One of the most physically destructive viruses seen *in the wild* was the `CIH` virus (also known as *Chernobyl*), first detected in June 1998 [72]. This virus replicated within Windows 95 executable files, and possessed a payload which would overwrite the hard-disk's boot sector, as well as corrupting the computer's Basic Input/Output System (BIOS) Read Only Memory (ROM) (when executing on susceptible machines) on a trigger date of April 26th (the anniversary of the 1986 accident at the Chernobyl nuclear power plant, in the Soviet Union).

Macro viruses (first observed around 1996) [73] exploited the macro scripting language embedded within popular office applications such as Microsoft Word; and by 1999 had been exploited in the wild, with examples such as *Melissa* spreading via email attachments [74]. (Although generally regarded as viruses, this use of networking to propagate blurs the lines between viruses and worms. See Section 2.2.2).

With the prevalence of anti-virus software on Windows systems, as well as changes to the OS itself (and the growth of alternative, better secured, OS such as Linux and MacOS) viruses have substantially diminished as a real-world threat in modern times.

2.2.2 Worms

Worms are self-replicating programs which spread to other computer systems via network connections. The first worm, known as *Creeper* was written as an experiment in 1971 [75], and ran on PDP-10 mainframe computers connected to the Advanced Research Project Agency Network (ARPANET). The *Morris Worm*, the first worm to cause significant disruption, was released in 1988 and was the trigger for the instigation of security on Internet connected systems, which had not been previously widespread [76]. It triggered what can be viewed as the first Denial of Service (DoS) attack on the Internet, although sources disagree as to the intent of the worm [77] [78].

A more destructive worm, was 2001's *Code Red* [79]. Whereas the Morris worm had spread to largely unsecured systems, using *sendmail*, Code Red spread via HTTP, exploiting a vulnerability in Microsoft's Internet Information Services (IIS). Code Red was estimated to have caused more than $2.6 billion worth of damage. [80].

Worm-like malware spreading via networked computers remains a major security problem today. The *Petya* and *NotPetya* malware [81] which caused more than $10 billion in total damage in 2017 [82], spread in part by exploiting a vulnerability in Microsoft Windows [83], known as *EternalBlue* [84].

2.2.3 Trojans

Named by analogy with the *Trojan Horse* of classical literature (*"Timeo Danaos et dona ferentes"*), a software *trojan* is a piece of software designed to trick users into installing it onto their systems, by virtue of masquerading as some other (legitimate) software. Pretty much all modern malware is encompassed in the form of a trojan, since modern OS configuration models generally restrict the access of software which is not running with *root* or *administrator* permissions, and simply asking the user to provide such execution permissions (by tricking them into thinking they're running some other legitimate piece of software) obviates the requirement to find and exploit *privilege escalation* vulnerabilities. Trojans are generally not self-replicating, but rely on users being duped into downloading and installing them via social engineering techniques (often disguising the file as a driver or system tool, such as Flash player [85]), or via *malvertising* (where the malware is embedded in code contained within advertising on a legitimate web page [86]). Since 2004 trojan malware has been shown to be a threat to mobile devices [87], in addition to more traditional computing platforms.

2.3 Attack types

Although most modern malware is of a hybrid type (often combining elements of trojans and worms), its development and distribution is commonly regarded as akin to a business by its developers [88]. There are four main types of end-effect: ransomware, botnets, remote access & exfiltration, and denial of service attacks.

2.3.1 Cryptoransomware

One of the most serious forms of modern malware is cryptoransomware. Cryptoransomware is a class of malware which encrypts the contents of a target computer — denying the user access to, and use of, the machine and the data files stored on it unless they make a payment. Although originally targeting more technologically naïve individuals, the primary targets of cryptoransomware today are large organizations with out-dated, unpatched machines, or poor cyber-defences; with criminal gangs often obtaining large payouts from the target's cybersecurity insurance policies [89].

The phenomena of encrypting a user's files and extorting payment to restore them can be traced back to 1989 and the 'AIDS Information Trojan' [90], although perhaps the best known example is the *WannaCry* worm which caused significant disruption to the UK National Health Service (NHS) in May 2017 [91]. WannaCry exploited the same EternalBlue vulnerability as NotPetya to propagate [84]; despite a patch for this vulnerability having existed for some months by the time of the WannaCry attack [92].

Although generally thought of as attacking traditional computing devices, by 2016, concerns were already being expressed about the potential to attack mobile, and other non-traditional, devices [93].

2.3.2 Botnets

Botnets consist of large-scale distributed computing resources, formed from malware-compromised machines and devices, connected together over the Internet [94]. Botnets have been used as relays to send unwanted *spam* emails (including those used in *phishing* attacks — where users are tricked into opening email that either contains trojan malware, or which attempt to elicit and steal personal information or credentials); and as a means of triggering a Distributed Denial of Service (DDoS) attack [95]. More recently there has been a trend to use botnets as distributed computational engines, for tasks such as *mining* crypto-currencies [96] or password cracking [97].

2.3.3 Remote access and data exfiltration

Another commonly targeted effect of malware is to establish remote access to, and/or a data exfiltration path from, an attacked machine [98]. Such attacks can be used for both cyber espionage [99], as well as more common theft of personal or commercially sensitive data [100]. One of the first widely publicized Remote Access Trojans (RATs) was *Back Orifice* developed by Josh Buchbinder (also known as *Sir Dystic*) in 1998 [101], which exploited flaws in early Internet-connected versions of Microsoft Windows. The threat from RATs persists with modern examples such as *njRAT* and *DarkComet*. Tools such as these have been used to obtain private information [102], including being used against political dissidents and Non-Governmental Organizations (NGOs) [103].

2.3.4 Denial of Service attack

Denial of Service (DoS) attacks are a type of attack designed to disrupt the service provided by a system or server. Most commonly DoS attacks are staged by sending multiple requests to a node in an attempt to overwhelm its ability to respond [104]. The first DDoS attack was seen in 1999, and utilized a group of computers (what would today be called a botnet) consisting of 227 nodes [105]. By 2016, botnets spread by the *Mirai* malware exceeded 400,000 nodes in size, and were offered by their creators on commercial-like terms for rent [106].

2.3.5 Physical attacks against network infrastructure

Physical or electronic attacks may be adopted against network infrastructure or devices. These could take the form of denying use of Radio Frequency (RF) communications through the use of jamming or other types of electronic RF attack against the transceivers, or by physically attacking cable or fibre-optic network links. This type of attack may also be directed against end-point devices, in order to try to damage or destroy them.

These types of attacks could take a number of different forms, depending on the sophistication of the attacker and the degree to which they were prepared to be overt in mounting such attacks. At their simplest, these types of attack may involve an attempt to sever a network or power cable, using techniques ranging from manually cutting it through to the use of explosive ordinance. More advanced techniques such as jamming [107] or electronic attack (using specifically generated RF waveforms to disrupt or damage the sending or receiving equipment) [108] — including the use of directed energy weapons — require increasingly sophisticated equipment. As such, outside of the battlefield environment, this type of attack is most likely to be confined to highly-targeted attacks, directed against very specific high-value targets.

Mitigation against these threats requires physical security guarantees for the operating environment, and RF spectrum being used. As such, they are beyond the scope of a networking or software centric solution, and are therefore not considered further in this thesis.

2.4 Software vulnerabilities

Most malware will exploit vulnerabilities inadvertently included within OS, systems, or application software (or occasionally in hardware [109]) on a device or computer system. Various taxonomies of vulnerabilities exist [110], [111]; but amongst the most commonly exploited vulnerabilities [112]–[114] are Buffer Overflow and Code Injection vulnerabilities.

2.4.1 Buffer overflow

A buffer overflow is a situation whereby an *input buffer*, a local variable within a program stored on the computer's stack memory, is provided with data larger the the allocated space, and overwrites other data stored on the stack. By carefully crafting the data written to the buffer, it is possible to overwrite stored register values (such as the return address for a function call), causing the flow of program execution to be altered: for example spawning a shell to permit the execution of any other arbitrary code. The potential threat from buffer overflow was identified as early as 1972, in a report commissioned for the USAF [115]. The 1988 Morris worm (see 2.2.2) exploited a buffer overflow vulnerability in an implementation of the finger protocol [116]; but the technique was not widely exploited until 1996, when the iconic paper “Smashing The Stack For Fun And Profit” was published [117].

2.4.2 Code injection

Code injection vulnerabilities are caused when programs do not sufficiently *sanitize* input data, permitting carefully malformed input data to be parsed as program code [118], most commonly seen in the form of injecting Structured Query Language (SQL) code into online database applications. First described in 1998 [119], these attacks were widespread by 2006 [120], and SQL injection attacks represented around 25% of all attacks on websites in 2015 [121].

2.5 Software Libraries

As described in Section 1.2.1, much software today is built from smaller components, such as software libraries. In principle widely-used libraries should be free from vulnerabilities, since by having been exposed to many users and developers it is likely that any bugs or other errors present will have been identified and addressed. However this is not always the case.

A good example of a serious vulnerability being found in a widely-used open source software library is the 2014 *Heartbleed* [122] vulnerability (or more formally, CVE-2014-0160), found in OpenSSL. By exploiting *Hartbleed* an attacker could extract protected data (including material relating to cryptographic keys) from the memory of affected servers. The vulnerability itself was a buffer *over-read*; in which (similarly to the buffer overflow described in Section 2.4.1) the amount of data that the function in question operates on is mismatched compared to the size of the data buffer. In the case of an over-read, it becomes possible to read more data than the buffer contains — thus accessing the contents of memory which would otherwise not be available outside to an external process.

Heartbleed is an excellent example of the *assumption of trustworthiness* which can result from widespread use, where users implicitly trust in the validity of the software because of the assumption that scrutiny of the source code has already taken place. In the case of Heartbleed, the coding error had first been made in a release of OpenSSL several years prior to the bug being discovered, and it had been widely deployed to protect web-servers and other applications in that time despite the presence of the then-unknown vulnerability.

This example shows that there can still be serious vulnerabilities in widely adopted open source software, which can remain undetected for extended periods of time. Widespread use in itself should not, therefore, be taken as a guarantee of correctness.

However, in-part because the software's source code was readily available, developers were able to release (in just seven days) a fixed version for effected systems to install, thus ensuring that the fix was promulgated as rapidly as possible, and demonstrating the *open source software ecosystem* working effectively to address vulnerabilities.

This situation has been seen again very recently. In December 2021, a critical vulnerability — known as *Log4Shell* [123] (CVE-2021-44832) — was identified within an extremely widely-used logging library for Java applications. The exploitation of this vulnerability has the potential to lead to arbitrary remote code execution and DOS attacks. Again, patched versions were released very rapidly to address the vulnerability; although (as described in Section 2.3.1) just because a security fix has been made available, it does not mean that all users will have applied it. Vulnerabilities such as these may still be found in unpatched deployed systems.

2.6 Threats to IoT security

The proliferation of IoT devices on the market with little or no meaningful security associated with them [124] has shown the extent to which security is not regarded as a high-priority for IoT device designers.

In the UK, in 2018, HM Government's Department of Digital, Culture, Media & Sport (DCMS) published a "*Code of Practice for Consumer IoT Security*" [125] in conjunction with the UK National Cyber Security Centre (NCSC). This guide contained thirteen recommendations on how to design, build, and administer secure IoT systems (with the first three being key recommendations). This list in full consists of:

1. **No default passwords**
2. **Implement a vulnerability disclosure policy**
3. **Keep software updated**
4. Securely store credentials and security-sensitive data
5. Communicate securely
6. Minimize exposed attack surfaces
7. Ensure software integrity
8. Ensure that personal data is protected
9. Make systems resilient to outages
10. Monitor system telemetry data
11. Make it easy for consumers to delete personal data
12. Make installation and maintenance of devices easy
13. Validate input data

The need for such guidelines are clear, given the numerous examples of poor security practice within the commercial IoT, and the often significant real-world consequences [126] these have had. Even high-profile products, including connected cars from large multinational companies, have been shown to have been implemented without careful consideration of security — or in some cases without any security whatsoever [127]. Dragoni, Giaretta, and Mazzara identified the need for a culture of security within the IoT [124]; both from the perspective of the users of IoT devices, and device manufacturers.

IoT devices form a somewhat different target for malware and other attacks, than more general-purpose computing devices. Attacks against IoT devices can be categorized as having consequences on one of three areas of the operation of the wider system into which the IoT device is connected:

- Compromise of data or services from the devices
  Including DoS attacks against the service or the device
- Attacks against the device itself
  Either to cause damage to the device, or co-opt it as a node within a botnet, for attacking other targets — such as using it to participate in DDoS attacks
- Attacks against connected physical systems controlled by the device

2.6.1 Compromise of data or services

For classes of IoT device which record, or otherwise have access to, personal or sensitive data the most significant threat is that of compromise of that data. This type of attack may be against unsecured (or insufficiently secured) data connections to and from the device or wider system, or may be as a result of a compromise of the device itself via malware to exfiltrate sensitive data over the device's existing network connections [128].

2.6.2 Attacks against the device

Malware targeting the device, may seek to damage or disrupt the service from the device itself, or exploit the IoT device as a part of a botnet to use it for unauthorized purposes, such as mounting a DDoS attack.

2.6.2.1 Denial of use of the device

In this class of attack the intended target is the device itself, and the attack seeks to disrupt, destroy, or otherwise prevent the use of the device: either temporarily or permanently. Malware such as BrickerBot [129] is designed to inflict a permanent denial of use of a device, by corrupting the device's firmware or file-system leaving it in an inoperable (or *bricked*) state. Proof-of-concept attacks have also been described [130] which could permanently brick smart lightbulbs — jumping between networks by abusing point-to-point radio links.

2.6.2.2 IoT device ransomware

The types of cryptoransomware attack described in 2.3.1 are often not directly applicable to the IoT, since few IoT devices store significant quantities of critical data locally, and they do not typically have low-level access to the back-end cloud services. Ransomware attacks versus IoT devices are, however, a plausible threat. It is highly likely that a similar type of attack, designed to place the target device in a permanently unusable state unless a payment is made, will be seen against an IoT device in the foreseeable future. This concept was described by Wüest, in a presentation on the threat of Ransomware in the IoT, at CRESTcon 2016 [131], and has been the subject of some concern recently [132], [133].

2.6.2.3 IoT botnets and Distributed Denial of Service attacks

With the proliferation of IoT devices seen over the last few years, these devices have been the targets of malware authors, particularly those wishing to create botnets with which to stage DDoS attacks (the generally low-powered nature of IoT devices makes them poor candidates for use as distributed grid computer nodes). These attacks typically exploit serious security vulnerabilities in the design of many IoT devices, such as hard-coded passwords, lack of security updates to the device software, and insecure network configuration (such as providing unsecured remote network terminal interfaces) [97].

Although (as described in Section 2.3.4) DDoS attacks can originate from conventional computing devices, perhaps the best known example of a highly disruptive large-scale DDoS attack in recent history was the result of the Mirai malware, which targeted IoT devices to exploit their network connectivity. Devices infected with Mirai formed botnets which were used in October 2016 to target Domain Name System (DNS) servers run by Dyn — and which resulted in considerable disruption to the Internet during the attack: at one point exceeding 1.2 Tbit/s of network traffic, and consisting of an estimated 100,000 nodes [134].

Mirai operates by targeting unsecured IoT devices, especially those with hardcoded user authentication credentials [126], seeking to gain administrative permissions. Once infected the devices will respond to C2 instructions to join DDoS attacks [135].

2.6.2.4 Replay attack

A replay attack [136] is a type of attack where a valid message from one device within a system to another (such as a command to operate a connected system, or the value of a sensor reading) is *captured* by a malicious party, and is then *replayed* at a future time. For example if a message to trigger a device to reboot could be successfully captured and replayed, then an attacker could easily stage a denial of service attack against that device by constantly replaying the message. Similarly if a message consisting of a sensor reading could be replayed, then an attacker could cause damage or disruption to the system by replaying that (e.g. reporting that a temperature reading is normal, when in fact overheating is occurring).

2.6.3 Attacks against connected physical systems

In the context of connected CPS and IIoT devices, a third target for malware is the physical system or equipment being controlled by the device. This type of attack seeks to cause disruption or damage to a CPS connected to a networked device. Although famously an attack against a disconnected system, Stuxnet [137] is a an excellent example of how malware may cause physical damage to a CPS by changing the control parameters of the connected device or system. In the case of Stuxnet, the target was believed to be Iranian nuclear centrifuges in the Natanz plant [138], with the malware seeking to induce excessive forces in the centrifuges by overriding the control of the rotor speed and disabling safety systems [139].

Another attack type, known as Manipulation of Demand via Internet of Things (MaDIoT) has been recently identified [140]. In a MaDIoT attack, the power-grid itself is targeted by using IoT-controlled high-load devices (such as water heaters or air conditioning units) to rapidly switch in coordinated pulses, causing power grid frequency instability and power blackouts [141].

2.6.4 Supply chain attacks

One class of attack that may be seen versus military IoT or other Critical National Infrastructure targets, is compromise of the supply-chain through the use of counterfeit or malicious hardware [142]. Due to the complexity, timescales and cost of mounting such an attack, and the difficulty in targeting or controlling the attack, it is unlikely that this type of attack would be mounted against civil devices. However, the small production volumes, and specialist operating environments of military systems potentially leaves them open to this type of attack from hostile nation-states. Attacks such as this can be very difficult to detect, requiring specialist anomaly detection techniques [143].

Software supply-chain attacks [144] are a type of attack where malicious code is injected into a system during its development, by exploiting the widespread use of *package management systems* by software developers. This type of attack can be staged, either attacking the distribution channels for software packages, or by *typosquatting* and uploading malicious code packages with similar names to popular packages (such as misspellings or other typos) [145] for popular languages (such as JavaScript or Python). If successful, this type of attack can cause the widespread compromise of both IoT devices and more general-purpose computing devices, easily embedding arbitrary code within the targets.

2.7 Cyber threat actors

The National Cyber Security Centre of the Netherlands publish an annual cyber security assessment [146]. In 2017, they identified the following threat actors:

- Professional Criminals
- State Actors
  - Agents or operatives of nation-states involved in espionage and / or disruption activities.
- Terrorists
- Cyber Vandals and Script Kiddies
  - Cyber vandals are defined as individuals who *"... carry out cyber attacks as pranks, as a challenge, or to demonstrate their own capabilities."* [146]; whilst script kiddies (also known as *skiddies*) are *"... less-experienced intruders who depend on more knowledgeable crackers to automate attacks ... "* [147].
- Hacktivists
  - Individuals or groups who *"... carry out digital attacks for ideological or activism reasons."* [146]
- Internal Actors
  - Also known as *insider threats* these are individuals operating from within an organization for personal motivations (such as revenge); as well as the unintended effects of the actions of careless users. In the context of domestic IoT applications, internal actors can include intimate partners for those in controlling or abusive relationships [148].
- Private Organizations

Whilst the IoT is at risk from any of these actors, incidents utilizing malware such as Mirai show that perhaps the most direct threat is from professional criminals, hacktivists, and script kiddies. As CPS increasingly become Internet-connected, the potential threat to industrial plant and CNI from terrorists and state actors will only increase.

2.8 Encryption

Encryption refers to the reversible transformation of data or messages such that they may only be read (decrypted) by the use of a suitable decryption key, thus preventing their interception by unauthorized parties.

2.8.1 Historical ciphers

Although a complete discussion on the history of techniques for the encipherment of text is beyond the scope of this work, there are two historical techniques that bear brief description here: substitution ciphers and one-time pads.

2.8.1.1 Substitution ciphers

Most encipherment conducted before the 20th Century employed a variation on a substitution cipher. [149] The simplest such cipher system, known as the Caesar cipher, dates back to Ancient Rome, and relies on a simple transposition where each letter in the original message (known as the *plain text*) is moved n positions forward in the alphabet (wrapping around after z), and thus the key k is simply the value of n. An example is shown in Figure 2.1.

$$\begin{aligned} k &= 3 \\ plain &= \texttt{abcdefghijklmnopqrstuvwxyz} \\ CIPHER &= \texttt{DEFGHIJKLMNOPQRSTUVWXYZABC} \\ \\ p &= \texttt{hello world} \\ c &= \texttt{KHOOR ZRUOG} \end{aligned}$$

FIGURE 2.1: An example of encrypting a short message by using a simple transposition cipher

However, such transpositional ciphers offer almost no security whatsoever, due to the trivial number of potential keys.

Substitution ciphers may also use non-sequential substitution alphabets, and so dramatically increase the number of potentially valid keys to $26! = 4.033 \times 10^{26}$. This is still however trivially small in terms of the processing power of modern computers — and the search-space can be dramatically reduced by using techniques such as frequency analysis [149]. Additionally, when using this technique, the key becomes much more complex and requires separate distribution to all parties to the messages.

This challenge of secure key distribution was one of the major factors behind the use of pseudo-randomizing mechanical encryption devices such as Enigma, Lorenz, and TypeX, in the 1930s and through the Second World War [150]. Many of these machines were of course, as history now tells us, flawed in design and in operations — leading to the widespread decipherment of German military signals traffic by the Government Code and Cipher School (GC&CS): the historical predecessor to the modern-day Government Communications Headquarters (GCHQ), at the now famous Bletchly Park. [151]

2.8.1.2 One-time pads

An alternative approach for secure messaging is to use a *one-time pad*. When correctly using such a technique a message may be, from a mathematical perspective, perfectly protected from an eavesdropper [149]. Such a pad is a set of truly random data, of the same length as the message, held in common between the sender and the receiver, and (critically to the technique's security) each pad may be used one-time only.

In using a one-time pad, each bit in the message is transformed by applying an Exclusive OR (XOR) function (a Boolean function which returns a 1 if the two input bits are different, and a 0 if they are the same). The resulting cyphertext of the plain text message contains the same amount of entropy as the one-time pad, and therefore **if the data in the pad are truly uniformly distributed**, for a message m of length L — the corresponding cyphertext M is valid for all possible messages of length L: with an **equal probability**.

For example if m is "HELLO": then we can encode this using American Standard Code for Information Interchange (ASCII) as:
$m = 72, 69, 76, 76, 79$ in decimal, or
$m = 01001000, 01000101, 01001100, 01001100, 01001111$ in binary.

If the corresponding section of the one-time pad, p is:
$p = 1001011010111110110001111010011010011000$

Then the cyphertext M would be: $M = m \oplus p$, as shown in Figure 2.2.

$$\begin{aligned} m &= 0100100001000101010011000100110001001111 \\ p &= 1001011010111110110001111010011010011000 \\ M &= 1101111111111011100010111110101111010111 \end{aligned}$$

FIGURE 2.2: An example of encrypting a simple text message by using an exclusive-OR function, and a randomly generated one-time pad

However, for a truly random pad, p — all possible values for p are equally likely: so without knowing p; it would be equally valid to assume
$p = p_{guess} = 0001110000001100000010110000100100011101$

And hence that m_{guess} = 'TIGER'; as shown in Figure 2.3.

Although such a system is unbreakable — it is highly impractical for real-world use, since in essence it transforms the problem of securely delivering messages to a recipient, to one of securely delivering shared secret one-time pads of equal length to the message.

$$
\begin{aligned}
M &= 0100100001000101010011000100110001001111\\
p_{guess} &= 0001110000001100000010110000100100011101\\
m_{guess} &= 0101010001001001010001110100010101010010
\end{aligned}
$$

FIGURE 2.3: An example showing that an incorrect guess as to the value of the one-time pad will result in the erroneous decryption of the message

2.8.2 Asymmetric encryption

Asymmetric or public-key cryptography was first proposed by James H. Ellis and Clifford C. Cocks in 1970 and 1973 [152], [153] in what were at the time SECRET (and were subsequently declassified) reports; and independently re-invented as Rivest-Shamir-Adleman (RSA) encryption [154] in 1978.

This technique offers a solution to the problem of key distribution by separating cryptographic keys into a public and private portion. This simplifies the problem into one of exchanging the non-secret public keys; requiring only that the integrity, and hence the trust-worthiness, of those keys is protected. Their content, as implied by the name *public* does not need to be kept secret.

Asymmetric cryptography can be used in two main ways:

1. To encrypt a message using the recipient's public key. Only the holder of the private key can decrypt it and so read the plain text.
2. To sign a message using the signer's private key. Anyone with access to the corresponding public key can verify that the message was signed by the signer, and that it has not been altered since signing.

Typically in implementation, an algorithm such as RSA is used indirectly to encrypt a symmetric key that is used to encrypt the actual data; or to sign a secure hash of the data, rather than encrypting or signing the full data itself. (See Section 3.2.1).

Although perhaps the best known example, RSA is not the only widely-used asymmetric encryption algorithm, and cryptography based on the mathematics of elliptic curves [155] [156] has taken over in many applications because of its computational efficiency [157].

The asymmetric RSA cryptographic algorithm is generally well regarded as being secure [158] and is used to protect much secure Internet traffic. This was originally via the use of Secure Sockets Layer (SSL), although this is now regarded as obsolete due to security vulnerabilities [159], and today is replaced by use of the more recent TLS [160] (which is still regarded as secure). There is however still a small risk of compromise to a *man-in-the-middle* attack, if the key distribution is poorly engineered. If a malicious third-party is able to intercept the key exchange messages, it would be possible for that party to provide its own certificates in place of the genuine certificates in order to impersonate the server to the device, and the device to the server. Although such a party never has access to any of the private keys it could still successfully trick the device into thinking its communicating with the genuine server, since without additional measures such as key signing, the device would have no way to know that the public key retrieved did not belong to the genuine server.

2.8.3 Diffie-Hellman key exchange

A related technology, is a technique to enable the generation of a shared-secret (to be used for generation of a shared secret key), by exchange of public messages. This was also first invented within GCHQ in the 1970s (by mathematician Malcolm Williamson[1]), and which was independently invented by W. Diffie and M. Hellman in 1976 [161]. This technique (commonly referred to as Diffie-Hellman key exchange) enables the parties to exchange data *in public* which can be used to generate a common shared-secret which can then be used as the basis of a cryptographic key for use with conventional symmetric encryption techniques. The original Diffie-Hellman algorithm uses exponentiation and modulo division (performed using values selected in secret by the two-parties to the exchange, and using publicly agreed values for a modulus and *generator*); but Elliptic Curve Diffie-Hellman (ECDH) which uses Elliptic Curve Cryptography (ECC) techniques is now commonly used.

2.8.4 Quantum computing

Although very widely used today, algorithms such as RSA or ECC are vulnerable to the power of quantum computing [162]. A quantum computer is a computational device that uses the effects of quantum mechanics, such as superposition, to perform calculations [163]. By using a quantum computer of a sufficient scale, and techniques such as *Shor's algorithm* [164], computations such as integer factorization could be performed fast enough to break today's cryptography [165]. Although no known quantum computer is able to operate at this scale today, the development of cryptographic techniques (such as lattice encryption [166]), which are resistant to known quantum computing algorithms, have been underway for some time.

2.9 Summary

This Chapter has explored the background of cybersecurity and encryption for the Internet of Things; examining a history of malware, and a number of common attack types and

[1] https://www.gchq.gov.uk/person/malcolm-williamson

threat actors especially as they apply to the IoT. These topics will be revisited in Chapter 8, where solutions to many of the problems described here will be proposed in the context of this research.

The next Chapter will explore some common Internet protocols, and examine in detail some specifically related to the IoT.

Chapter 3

Commonly used Internet of Things Protocols

This chapter will examine the types of communications protocol used for general-purpose Internet traffic, as well as looking in detail at some specific protocols designed for limited-bandwidth applications such as may be encountered within the IoT: in particular the MQTT protocol which has been used extensively within this research. It will also describe some architectural models that can utilize these protocols to provide interfaces to IoT devices, and look at some of the communications architectures in use today within the IoT.

3.1 Internet communications

Any Internet-connected device, whether a traditional computing device or an IoT device, requires suitable network protocols to enable it to send and receive data.

The Internet Protocol Suite (TCP/IP) [167] is organized into four abstraction layers:

1. Link Layer
   Defining the lowest-level of the stack, and considering the physical bearer and associated networking standards: such as Ethernet (IEEE 802.3) [168] or WiFi (IEEE 802.11) [169]
2. Internet Layer
   Providing inter-networking protocols: e.g. Internet Protocol Version 4 (IPv4) & IPv6 [30]
3. Transport Layer
   Providing host-to-host communications, such as Transmission Control Protocol (TCP) & User Datagram Protocol (UDP).

4. Application Layer
   Providing the protocols directly used by applications — such as HTTP, File Transfer Protocol (FTP), Secure Shell (SSH), & MQTT.

3.1.1 Link and inter-networking layer protocols

Ethernet [168] was originally developed by Xerox at their PARC facility in the 1970s, and is based on the earlier AlohaNet system [170], and which was commercialized and standardized in the 1980s [171]. Today it is an almost ubiquitous link-layer protocol for wired communications, outside of specialist protocols used within applications such as HPC. Whilst the original implementation had a speed of just under 3 Mbit/s [172], modern *Terabit Ethernet* [173] can reach speeds of up-to 400 Gbit/s [174].

The Internet Protocol (IP) is a protocol for routing data packets across network boundaries (or internetworking). Its use has become widespread since its development by V. Cerf and R. Kahn in 1974 [175] as a part of the DoD funded ARPANET [176]. Although other protocols, such as Asynchronous Transfer Mode (ATM) [177], have been used historically, today IP based networks have become ubiquitous in most applications.

3.1.2 Transport layer protocols

Traffic on IP networks is carried as either TCP or UDP packets. TCP [175], [178] provides *reliable* (assured delivery) communications with error checking, and in-order delivery. By contrast, UDP [179] provides connectionless communications, where data is sent with no guarantees of delivery or the order of that delivery.

3.1.3 Application layer protocols

Since the inception of the prototypical form of the ARPANET, in the late 1960s, numerous Application Layer communications protocols have been used to carry different types of information over the network. Some, such as Gopher [180] have faded into obscurity, whilst others, most notably HTTP [21] have become near ubiquitous since its invention and the development of the World Wide Web in 1990 [181].

Although the use of HTTP is very widespread, it is an unencrypted protocol; thus any data sent using HTTP is open to interception by third-parties. Given the requirement for secured communications, an encrypted connection is therefore required. (See Section 3.2.3).

3.2 Cryptographic protocols

3.2.1 Modern symmetric encryption and message digest algorithms

The most common way to secure information in-transit over the Internet is by using Transport Layer Security. The Secure Hyper-Text Transfer Protocol (HTTPS) uses TLS to both facilitate the encryption of the data traffic, and to provide assurance around the authenticity of the web server supplying the data.

The data itself is commonly encrypted using the Advanced Encryption Standard (AES) algorithm, using a 256-bit key.

AES was adopted by the United States' National Institute of Standards and Technology (NIST) in 2001 [182], using a cryptographic scheme proposed by Joan Demen and Vincent Rijmen, named Rijndael [183].

NIST also standardized the Secure Hash Algorithm, version 2 (SHA-2) cryptographic hash functions [184] in 2002 [185]. Cryptographic hash algorithms provide a one-way operation that will generate a fixed-length output for any given input. The generated hash value is unique for any given input, and as such provides a simple and computationally inexpensive mechanism to determine if any two blocks of data are identical or not [184].

The 256-byte SHA-2 (SHA-256) hash algorithm (which produces a 256-byte digest), is generally used in combination with RSA for message signing.

3.2.2 IPSec

One approach to the provision of secure communications, is to secure the whole IP stack — using Internet Protocol Security (IPSec) [186].

IPSec encrypts the entire communications stack at the Internet Layer — meaning that applications sending data over an IPSec protected connection do not need to be aware of the security (unlike TLS which runs at the transport layer). IPSec is most commonly used to provide very secure end-to-end encryption for high security Virtual Private Networks (VPNs). IPSec is compatible with IPv6 and mechanisms for its efficient use over 6LoWPAN have been identified [187]. IPSec is however relatively uncommonly used on public communications networks — outside of its use for VPNs and Voice over Internet Protocol (VoIP).

3.2.3 Transport Layer Security

The alternative to encrypting the whole stack, is to use a protocol to secure the data. Transport Layer Security 1.0 [188] was the successor to the original SSL [189] protocols used to secure Internet traffic, and has been updated since its inception to the (now) widely supported TLS 1.2 [160] and the most-recent TLS version 1.3 [23] (published in 2018).

TLS facilitates HTTPS [190], providing both security for the connection by encrypting the data, and authentication of the identity of the communicating parties. In the context of a website visited using HTTPS establishing the identity of the web server is especially important — since this effectively prevents genuine sites being spoofed via a *man-in-the-middle* attack.

A TLS connection is established via a *TLS handshake* [160]. The client device attempts a connection to the server (the `CLIENT HELLO` message), which describes the version of TLS that it supports, and which of the various *cipher suites* that it supports. The server then responds (with a `SERVER HELLO` message), which indicates which of the cipher suites the server has selected for use, and a copy of the server's certificate. Typically a modern client and server will use either Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) or RSA to facilitate key-exchange and deriving a 256-bit key to be used with AES to encrypt the traffic (commonly operating in *Galois/Counter Mode* (GCM) to permit higher throughput), and using RSA together with SHA-2 with a 256 or 384-byte digest for message signing. (e.g. using the `ECDHE-RSA-AES256-GCM-SHA384` or `AES256-GCM-SHA384` cipher suites).

HTTPS uses TLS to encrypt the HTTP connection [190] and is very widely used to secure Internet traffic. Although originally reserved for use with specifically secure web sites (such as Internet banking or shopping), the use of HTTPS is now increasingly widespread as concerns grow over privacy and the interception of personal information, and as the barrier to entry to use TLS is lowered through the provision of free Certificate Authority providers such as *Let's Encrypt* [191].

In addition to its use to encrypt traffic, TLS also provides authentication of the identity of the web-server being visited, via the use of International Telecommunication Union X.509 certificates [192].

3.2.4 X.509: certificates and identity

Although formally defined by the International Telecommunication Union (ITU), X.509 certificates as they are generally used in practice on the Internet, are used as defined by the Internet Engineering Task Force's RFC5280: "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile" [193]. This defines a hierarchical scheme to establish a root-of-trust for identity on the Internet. For example, a website wishing to establish its identity can obtain a certificate from a trusted third-party known as a Certificate Authority (CA), which has been signed by that issuer. That CA possesses an intermediate certificate that has been signed by an intermediate CA, and ultimately that intermediate CA's certificate will have been signed by a root CA. The public keys associated with this small number of root CAs are typically distributed as a part of the operating system or web browser application software, and hence may generally be assumed as being already present on the target device.

The system is also sufficiently flexible to permit anyone to generate a root certificate, and a user who trusts the issuer of this certificate may use it to form their own root-of-trust in determining the veracity of other certificates signed by that root. This means that a company or organization can issue their own chain of certificates, without recourse to any external organization, for use in establishing the identity of servers or devices.

Figure 3.1 illustrates the process of certificate generation.

X.509 uses a Certificate Signing Request (CSR) to enable the generation of a certificate. This not-only enables the generation of a certificate corresponding to a key-pair, without exposing that key-pair to the signing CA directly, it also permits the incorporation of a number of fields into the certificate. Of particular importance is the subject field which provides several attributes, including the Common Name (CN) which is used (in the case of certificates for use with TLS) to denote the Universal Resource Locator (URL) that the certificate applies to.

3.2.5 Quick UDP Internet Connections

Quick UDP Internet Connections (QUIC) is an experimental secure internet transport layer, first proposed by Jim Roskind from Google, in 2012 [194]. QUIC provides a multiplexed connection over UDP, in order to provide TCP-like connection with reduced latency and improved performance for loading pages, something observed in real-world testing [195]. QUIC utilizes TLS 1.3 and requires all connections to be encrypted using TLS using a minimum handshaking overhead [196].

Due to the influence of Google, and their ability to control both client (through the Chrome web browser) and server side (through their provision of many popular websites) — it was estimated in 2017 that QUIC accounted for around 6% of all Internet traffic [197].

3.2.6 Datagram Transport Layer Security

Datagram Transport Layer Security (DTLS) [198] is based on the operation of TLS, but uses *unreliable* UDP packets rather than TCP [199]. Although generally regarded as

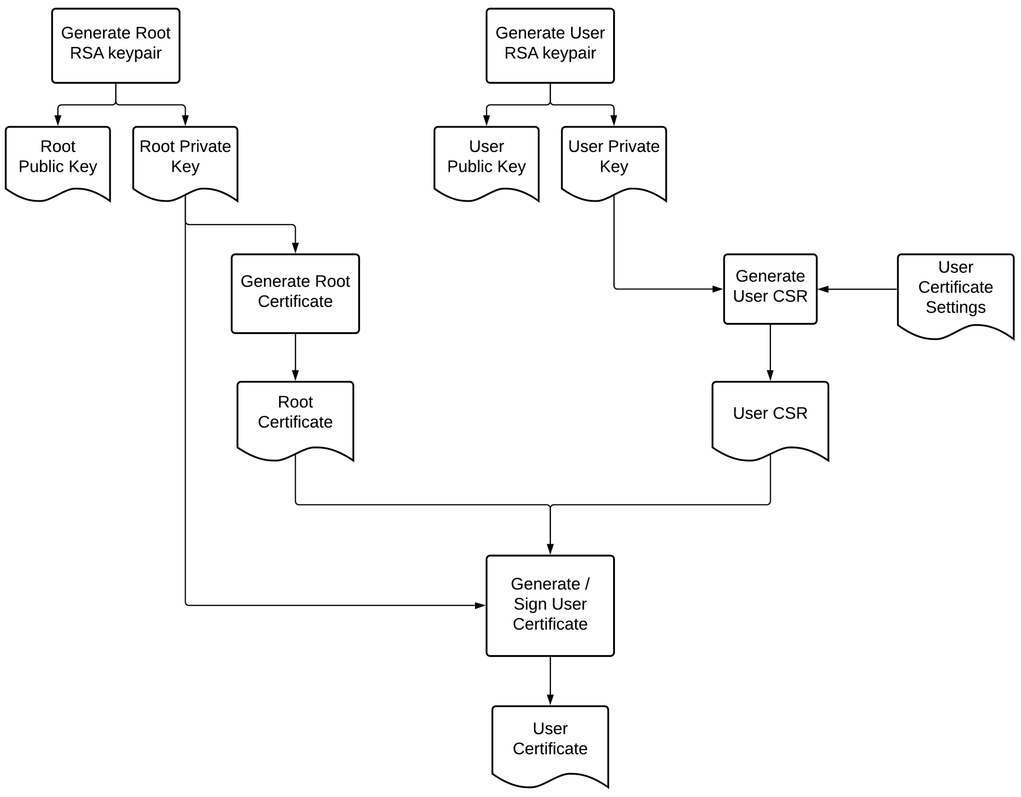

Generate Root RSA keypair
Root Public Key
Root Private Key
Generate Root Certificate
Root Certificate
Generate User RSA keypair
User Public Key
User Private Key
Generate User CSR
User Certificate Settings
User CSR
Generate / Sign User Certificate
User Certificate

FIGURE 3.1: A flowchart depicting the process to generate an X.509 certificate and CA certificate

secure, some plain text recovery attacks have been shown against some implementations of the protocol [200].

3.2.7 Secure shell

SSH [201] is a secure remote-access protocol widely used for remote command-line login and execution. SSH uses public-key cryptography (typically RSA) to authenticate the user and remote computer. This enables a user who requires frequent access to a remote server to store a copy of their public-key on the server (and associated with their user account on that server), and then use the presence of their private key on the device they are initiating the connection from as a determination of their identity rather than having to enter their password. For even greater security, use of a password to log in can be disabled on the server, preventing any user who does not possess a private key corresponding to a public key on the device from logging in.

SSH also provides authentication of the identity of the remote server. When establishing the connection to a server for the first time, the system will display an Elliptic Curve Digital Signature Algorithm (ECDSA) key fingerprint in the form of a SHA-256 hash. If accepted this will be stored (along with the target's Fully Qualified Domain Name (FQDN) and IP address). If subsequently the value received on connection differs from the stored version, the user will be warned as to the possibility of an *man-in-the-middle* attack.

The SSH protocol is also used as the basis for the SSH File Transfer Protocol (SFTP), commonly used for secure remote file copy.

3.3 Constrained protocols

As described in Section 1.2.3, deployed IoT devices may be expected to need to be able to make efficient use of poor quality network conditions. As such they can be expected to make use of protocols that are specifically tailored for such situations.

HTTP (and the secured HTTPS) is an ideal protocol to provide human computer interfaces. Due to the ubiquity of the use of HTTP for the web, just about any conceivable general-purpose computing device will have at least one web browser application available, and a human operator could therefore use this to interact with an IoT system. Furthermore if a bespoke local application is desired — just about every major programming framework will include support for HTTP / HTTPS. HTTP is also well suited to situations where you wish to expose an Application Program Interface (API) to other applications. Using a Representational State Transfer (REST) [202] API over HTTP is very widely supported and it makes it easy for a third-party application to consume a service without any detailed knowledge of the implementation of the service.

However, for all its ubiquity in the desktop and handheld computing domain, HTTP has a considerable overhead in terms of the size of its headers, and in the relative complexity of the process of establishing connections. Whilst this makes it an easy protocol to use, it is not a protocol that is especially well suited to some aspects of M2M communication. In particular HTTP is especially poorly suited for use in situations where high-frequency short messages are exchanged — the type of data flow that may be generated by a frequently updating IoT sensor. In this situation the *cost* of the message headers far outweighs the size of the data being sent. Whilst this is unlikely to be an issue for a device using a modern network connection: for a device in a location where network speed is measured in Kilobits per second, this is quite wasteful, and significantly limits the traffic rate of the device.

To address this requirement, there are a number of *constrained* protocols optimized for use with IoT devices. These protocols are specifically designed to have a small data overhead such that they can be effectively used in situations where network bandwidth is limited — or connections are intermittent. The performance and efficiency benefits (in terms of overheads and constraints) for IoT protocols in comparison with HTTP, are described in detail by Naik [203].

Two of the most commonly used constrained protocols within the IoT [204], [205] are MQTT, and the Constrained Application Protocol (CoAP). These will be examined further in the following Sections.

3.3.1 CoAP

CoAP [206] — is a web transfer protocol designed for constrained environments. It provides a RESTful *HTTP-like* request/response paradigm — but unlike HTTP it uses very small data headers designed for use in low-bandwidth (and limited computing power and memory) scenarios such as may be found within the IoT [206].

When using HTTP to connect an IoT device to the Internet, the device would typically be the client — and would connect over HTTP to a web server on the Internet to retrieve (`GET`) or send (`POST`) data. However, with CoAP this is reversed. The typical implementation would have the IoT device as the CoAP server — and Internet resources acting as the client to retrieve or send data, to and from the device.

CoAP uses the connectionless UDP [179] as the transport layer. CoAP also supports Group Communication via IP Multicast, enabling the efficient sending of data to multiple recipients [207].

Within the CoAP standard, a mechanism to enable resource discovery is specified. This enables a client to request a list of all of the resources that a server provides, by visiting a standard CoAP Uniform Resource Identifier (URI) on that server.

```
coap://example.com/.well-known/core
```

The list returned includes metadata about the resources, all specified using the Constrained RESTful Environments (CoRE) link format [208].

CoAP can be used in a secure mode — where DTLS is used to encrypt the underlying UDP traffic [209].

3.3.1.1 CoAP Publish and Subscribe

CoAP supports a publish-subscribe model, known as Resource Observe — where a server can track observers, and notify them when the end-point they are observing is updated; but this is un-brokered and so requires the server to actively track the subscribing observers.

3.3.1.2 CoAP Quality of Service

CoAP supports two levels of Quality of Service (QoS). Requests and responses may be either confirmable (in which case they must be acknowledged by the receiver) or non-confirmable. Neither CoAP nor the underlying UDP transport layer provides any guarantee about the delivery of non-confirmable messages.

3.3.1.3 Network access

Given the server-oriented nature of a typical CoAP device, in order for a CoAP client to establish a connection to a CoAP server, the client must be able to route to it. This is not typically possible in environments where NAT is in operation without specialist setups, tunnelling, or (for simple cases with only a single device on the local network) port forwarding. As such CoAP is better suited to networking environments using IPv6 — which permit direct routing to the device from the Internet.

CoAP is still relatively immature and is not yet very widely used, although there are a number of reference implementations available for a number of different programming languages and development platforms: for example, the CoAPthon project provides an easy to use Python implementation, but it has been shown to have relatively poor performance under high load conditions [210].

3.3.2 MQTT

MQTT is a lightweight brokered publish / subscribe messaging protocol, originally developed by Andy Stanford-Clark (IBM) and Arlen Nipper (Arcom) in 1999 to provide lightweight telemetry for the oil and gas industry, and it became an OASIS standard in 2013 [24]. Version 5.0 of the MQTT standard was published in 2019 [211].

MQTT runs over TCP/IP networks, using TCP [178] as the transport layer. Unlike CoAP and HTTP, all messages must be routed via a broker. The broker is responsible for tracking all subscriptions, and sending data to subscribers when a publisher issues a message. There are a number of open-source brokers — two of the best supported are Mosquitto and Paho — both *Eclipse IoT* projects.

This research makes extensive use of MQTT, and the Mosquitto broker [212], [213] has been used throughout.

3.3.3 MQTT Messages

All data exchanges using MQTT take the form of *messages*. Messages consist of the MQTT message headers, and a user-defined block of variable-length data, known as the *payload*. The MQTT specification makes no attempt to define the meaning of the payload, and in fact explicitly declares that *“ The content and format of the data is application* *specific."* [211]. This means that applications using MQTT are free to define their message payload as required.

Messages are disseminated by being sent to a broker, using a particular topic. The broker will respond by relaying the message payload to any subscribers that have registered a subscription for that given topic.

3.3.3.1 MQTT Topics

Under MQTT all messages belong to a topic and connected clients can subscribe or publish to any topics they have access to. Topics in MQTT are hierarchical, so a client subscribing to a base topic, can also elect to receive any messages published to any of its subtopics using wildcards.

For the purposes of subscriptions, topic wildcards are supported via the `#` and `+` characters, with `#` denoting a *multi-level wildcard*, and `+` representing a *single-level wildcard*.

This is described further in the standard description, with the following examples. A subscription to `sport/tennis/#` will receive all messages sent to any of `sport/tennis/player1`, `sport/tennis/player1/ranking`, or `sport/tennis/player2`. Whereas a subscription to `sport/tennis/+` will receive `sport/tennis/player1`, and `sport/tennis/player2`, but NOT `sport/tennis/player1/ranking`. The single-level wildcard can also be used to specify a subscription topic in the form: `sport/+/player1`, which would receive both `sport/tennis/player1` and `sport/golf/player1`. [211].

3.3.3.2 MQTT Quality of Service

MQTT supports three levels of Quality of Service, ranging from the *fire and forget* nature of QoS0 (*"Deliver at most once"*), where there are no delivery guarantees beyond those offered by TCP, to QoS1 (*"Deliver at least once"*), and QoS2 (*"Deliver exactly once"*), where the receipt of each message is actively confirmed. QoS2 is correspondingly the slowest QoS level — and is therefore generally only used for situations where reception of duplicate messages would be actively harmful. MQTT supports multiple different QoS for different connections (i.e. a given device could publish on one topic using QoS1, and subscribe on another topic at QoS0).

3.3.4 MQTT Security

MQTT is by default an insecure protocol (all data is sent as plain text), and has no authentication (anyone can connect to a broker, and publish or subscribe to any topic). In this configuration, the protocol is clearly unsuited for use in any secure applications, however it can be secured by the application of TLS to the message traffic, and the use of authentication to restrict access to the broker. If using encryption, it is clearly necessary to also use some form of access control, otherwise an eavesdropper could simply request a subscription to the topic of interest.

Mosquitto supports both conventional asymmetric (public-key) encryption over TLS, and the use of Transport Layer Security using a Pre-Shared Key (TLS-PSK) [214]. TLS-PSK is not widely adopted but it has some potential advantages over the use of asymmetric keys in the context of very low powered IoT devices. As the level of processing required is reduced, it is better suited for use in systems with very low-powered processors. Not all MQTT clients support TLS-PSK however, and the sample Mosquitto clients provide the most complete implementation. The price of using this method is that it requires the system to define a secure method for private key exchange, and that it provides no *future security* [215] (so in the event of a compromise of the key, all previous messages protected using that key are compromised). As the power of IoT devices increases the value of using this approach over conventional TLS reduces.

Further detail on the MQTT protocol can be found in Appendix A.

3.4 Summary

This Chapter has examined a number of Internet protocols, and in particular the MQTT protocol which is very widely used within the IoT; and which is the basis of this research work. It has also described the use of Transport Layer Security and X.509 certificates in the context of secure Internet communications.

The next Chapter will introduce the final background concept to this work, and discuss the application of Command and Control architectures to IoT messaging.

Chapter 4

Command and Control Architecture and the Internet of Things

The previous Chapter explored some communications protocols and architectures for use within the IoT. This Chapter will examine the concept of Command and Control and some different architectures for C2 networks. It also considers the security requirements for C2 messages, and examines some C2 message types and their applications as they pertain to use within the IoT.

4.1 Command and Control

Command and Control (C2), whilst traditionally thought of in a military context, simply denotes the process whereby a hierarchically superior entity, sets an objective for (command), or provides direct instructions to (control), a subordinate entity. More generally, it is defined by the North Atlantic Treaty Organization (NATO) as: *"... the exercise of authority and direction by a commander over assigned and attached [resources] in the accomplishment of [a goal]."* [216].

Within the context of this research, C2 can be thought of as referring to any communications which take place within an hierarchical structure where information is exchanged between a *controlling entity* and a *subordinate entity*, in order to enable direction of the operation of the subordinate entity by the superior controlling entity.

There are two main types of C2 architectures: a purely hierarchical approach, such as is typically adopted within the context of military or civilian law-enforcement personnel, and a peer-to-peer approach.

4.1.1 Hierarchical versus peer-to-peer architectures

Within a strictly hierarchical C2 model of communications, all communications between nodes must flow via a (common) superior commander.

In a military-context, it is typical to see multiple layers of command and control nodes — intermediate commanders responsible for making delegated decisions, and to pass reports up, and instructions down to superior and subordinate commanders.

Where there is a requirement for messages to be sent between *leaf nodes*, they will always be passed through a superior commander, typically the lowest level of command layer to which both nodes are subordinate. An example of this is shown in Figure 4.1.

Whilst not a perfect analogy, email may be thought of as something broadly operating along this type of C2 architecture. Users sharing a common mail-server will exchange messages within that server, but messages between users on different servers need to be passed over to a different server to be accessed by the user.

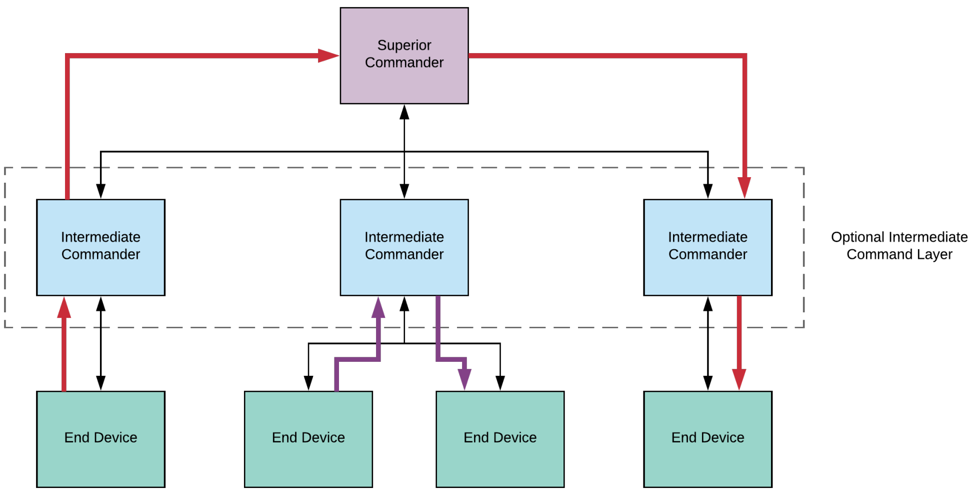


FIGURE 4.1: A diagram showing message flow between leaf nodes within a multi-level C2 hierarchy. The *purple* data flow goes through the common intermediate commander, whereas the *red* must go to the top-level commander since the two nodes involved share no other common command node.

By contrast to this approach, a purely peer-to-peer communications model permits direct communication between any nodes. An architecture adopting this scheme will have an entirely flat hierarchy, where all nodes are equal and any communications flows directly between end-points. In practice there are relatively few such models adopted in mainstream communications architectures. Traditional two-way radio communications is perhaps the best example, although there are also some *privacy-oriented* communications tools such as *Shazzle*[1] and *Mesh*[2] which do adopt a true peer-to-peer messaging architecture.

More typically within *nominally peer-to-peer* communications (such as the telephone system, for example) messages are routed or relayed via a server, exchange, or other broker at one or more levels of hierarchical extraction from the user. This is still the case within most End-to-End Encrypted (E2EE) messaging services [217] such as *Signal*[3].

4.1.2 C2 and the IoT

This work primarily considers a use-case commonly seen within the IoT — where a particular class (or family of classes) of device are deployed, and use a C2 function provided by a centralized service. That service can support one or more users, each of whom would have access to one or more devices.

This model, utilizing a system with centralized control, and providing an interface to control connected devices via the internet, is one that has been adopted by a number of commercially available products such as Philips Hue smart light bulbs [218].

An example of this architectural approach is shown in Figure 4.2.

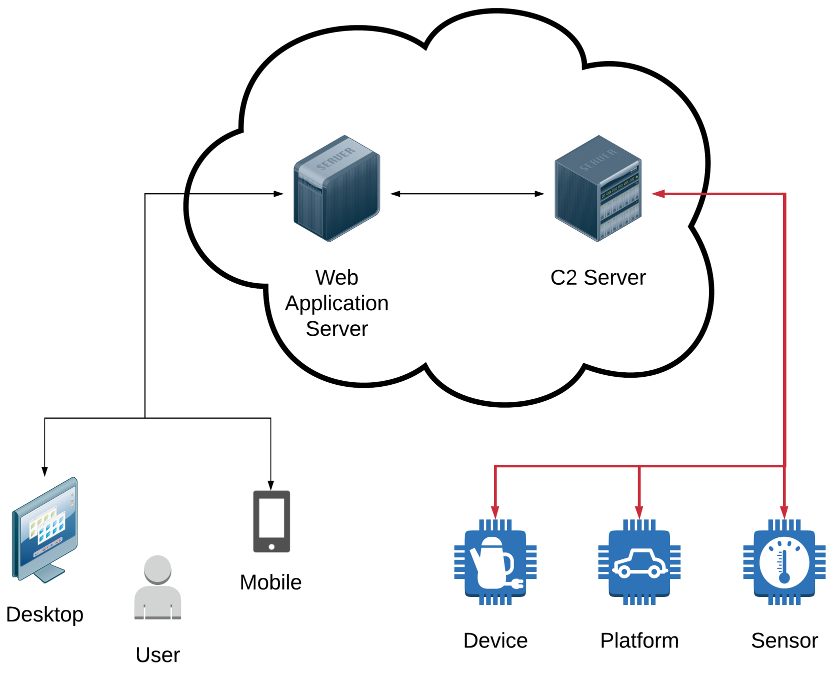


FIGURE 4.2: A diagram showing a generic C2 architecture for the Internet of Things.

[1] https://shazzle.com/shazzlechat/
[2] https://mesh.im
[3] https://signal.org

This is also compatible with the emerging trend to see IoT devices make use of, and integrate with, existing cloud services such as those provided by Google or Amazon. This approach provides easy integration with home-automation hub devices such as Amazon Echo (and the Alexa voice assistant) and Google Home. Typically the implementation of services enabling *smart* devices to be controlled by a voice assistant, will make use of a secured HTTPS API end-point. The voice assistant uses this REST API to exchange data with the smart device's C2 service. The C2 service then subsequently sending suitable messages to the end device to perform the actions requested by the user.

4.1.3 C2 for military applications of IoT

Given the requirement for IoT and IoBT systems in a military context (described in Chapter 1) there is a clear requirement for a C2 system to integrate them into existing military decision-making processes and structures. For example, the IoBT research conducted by the United States Army Research Laboratory proposes a future scenario where highly autonomous devices are ubiquitous in a battlefield environment [219], requiring a robust C2 system.

The infrastructure requirements underpinning network connectivity for a deployment of IoBT devices may be satisfied by deployed military systems, or the indigenous systems of the operational theatre (or a combination of both). However depending on the scale of any such deployment of IoBT devices, the infrastructure may be insufficient to meet the requirements for connectivity to support real-time C2. The use of meshed networks, consisting of device to device communication, and more complex network management may address some of these issues [52].

As described in Chapter 1, not all military use of IoT will be in the context of high-intensity war-fighting. Other military IoT will also require C2 for the devices — which for some applications may be required to integrate into existing military C2 systems via the automated generation of situational reporting. Raglin, Metu, Russell, *et al.* propose a "conceptual IoT to C2 framework" [220] that addresses aspects of this requirement.

Lastly it must be noted that the technical challenges of C2 messaging are distinct from the organizational challenges of decision-making, information management and situational awareness that future IoBT systems will also face [221].

4.2 Security requirements for C2 messaging

The traditional view of information security is one of *"Confidentiality, Integrity, & Availability. . . "* [222]; but within the IoT, this thesis proposes that this should instead be viewed as *"Confidentiality, Integrity, & **Authenticity**. . . "* [223] — since within the IoT availability may not be achievable, or (in the case of covert devices that might be employed in an IoBT context) desirable. Proof of the authenticity of the origin of C2 messages is critical within a C2 context — in addition to Integrity of the content of the messages themselves being assured. Whilst confidentiality will be a key requirement in many cases, it is not universally required — and providing a system can guarantee the other two, it may not always be necessary. This is especially true in situations where very low power consumption devices that are used for reporting on non-sensitive data and for which the additional power overhead of running encrypted communications may not be justified.

An illustration showing the classic triad redrawn for IoT is shown in Figure 4.3.

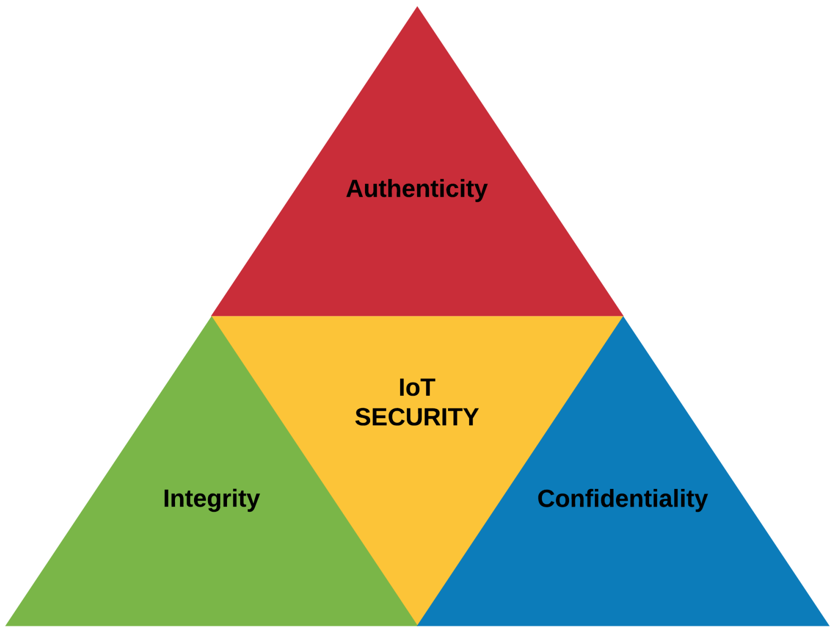


FIGURE 4.3: A diagram showing an alternative expansion of *CIA* for the Internet of Things.

These three elements are required for a C2 message to be considered secure:

1. Message Confidentiality
   Message confidentiality simply refers to the requirement that it not be possible for an external third-party to intercept and read a message in transit. This requirement is easily addressed by the addition of suitable message encryption.

2. Message Integrity
   Message integrity refers to the ability to verify that the message has not been changed or tampered with in transit, and to verify that the message has not been received before. This latter point is very important for IoT C2 traffic, since the ability to detect the re-transmission of a previously sent (but otherwise valid) message mitigates Replay Attacks (see Section 2.6.2.4). Without protecting against replay

attack, systems are susceptible to an attacker using traffic capture to store a previously sent message, and *replaying* that to perform operations on demand, or stage DoS attacks against the device. There are a number of different techniques to address this threat, which are discussed further in Section 8.1.

3. Message Authenticity
   Message authenticity refers to the requirement that it must be possible to verify that a given message genuinely originates from a specific sender. Solutions for both authenticity and integrity can be addressed using cryptographic signature techniques.

4.3 IoT Command and Control message types

There are three main types of message that are required within the context of an IoT C2 system:

1. Messages concerned with the operation of the devices
2. Messages concerned with the operation of the C2 network
3. Messages concerned with the operation of the functional system

4.3.1 Messages concerned with the operation of the devices

There are several types of message that form a broad class of messages which are required to carry any information required to change the state or operation of the device. Such messages can either be direct *command* actions (for example causing the device to switch on an effector, or enter a different power-mode), or may be *control* messages providing specific data (for example, a message specifying a new temperature value for a thermostat to locally regulate the temperature to).

4.3.1.1 Software update

One specialist type of device oriented C2 message, is a message pertaining to device configuration, and especially software update (which may be viewed as a special case of device configuration update). One of the major challenges in constructing a secure IoT system is the requirement to be able to perform secure remote updates of the configuration on the devices, since once devices are deployed the feasibility of performing in-person updates may be extremely limited.

Software in this context could include anything from device firmware for an embedded component, through to a set of configuration data. Although the term software is typically used to describe programs running on a device (in the form of binary executable or interpreted script code), a device may require other types of configuration data updates which may be considered alongside the traditional definition of software in this context. Such data may either be to change the operation of the device (such as specifying different operating parameters), or to provide system configuration data (for example, updating a cryptographic key).

A number of approaches have been adopted for this requirement, but they broadly fall into approaches focused on update of embedded firmware [224] or approaches more focused on the more general problem [225].

4.3.2 Messages concerned with the operation of the C2 network

A common C2 message type are those which are concerned with the operation of the C2 network itself. For example messages pertaining to the device's membership of the C2 network. Depending on the system these could include both requests from devices to join a given C2 network, and messages removing devices from the network.

4.3.3 Messages concerned with the operation of the functional system

Just about every IoT device will have a requirement to send or receive messages pertaining to the primary application of the system. For example, a device which is (or which contains) a sensor will almost certainly have a requirement to send the data from that sensor to another platform. Sensor data may need to be sent to a cloud service hosting a software application, or to an end-user device, or an external data store. Whilst this type of message may *strictly* be considered outside of the scope for a C2 message, any device operating in a C2-based system, could adopt the scheme devised for the secure transmission of C2 messages for these application messages, especially since (depending on the specifics of the system), there is a high likelihood that the same requirements for authenticity, integrity & confidentiality are present for application data as they are for C2 data.

With a desire to create a single, integrated messaging protocol to cover all of the messaging requirements of a system it is highly advantageous to include application data messages as a sub-type within the suite of C2 messages.

4.4 Summary

This Chapter has explored the concept of C2 messages, and how they relate to the IoT. This Chapter concludes the first Part of the thesis, which has focused on the background to this research. The second Part will start to explore the original research that has been conducted in order to answer the research questions. In particular, the next Chapter introduces the Secure Remote Update Protocol, a messaging protocol for the IoT developed as a part of this research, and which has been designed to provide a secure protocol for C2 messages in the context of IoT applications.

Part II

The Secure Remote Update Protocol

Chapter 5

The Secure Remote Update Protocol

This Chapter introduces and examines the first original element of this research: a protocol to address the requirements of secure C2 messaging for the IoT, and seeks to answer Research Question RQ1.

Elements of this Chapter have been previously published as [2] and [3].

5.1 Design concept

In order to address Research Question RQ1, the Secure Remote Update Protocol (SRUP) was designed to make use of well-established, and therefore well-known and well-trusted, components to implement a secure, efficient and straightforward protocol to carry C2 messages between IoT devices and a Command and Control server.

The protocol was originally envisaged as a mechanism to provide a means to signal deployed IoT devices to perform a software update (hence the name), but as research work progressed it was highly apparent that this same protocol could also be used as a general-purpose communications protocol for all C2 operations associated with an IoT device and its controlling server.

The SRUP protocol is built on top of MQTT (see Section 3.3.2) — and provides an efficiently packed binary message structure within the payload of an MQTT message. It uses cryptographic protocols such as SHA-2 and RSA to provide authentication and message integrity, and a system built using SRUP can also use TLS to provide message confidentiality, where required.

SRUP has been designed so that the specific cryptographic protocols in use, could be easily substituted independent of the C2 messaging protocol itself. Whilst the sender and receiver need to both be using the same cryptography in order to be able to successfully understand and verify messages, any given system utilizing SRUP could simply swap to a different implementation using any other asymmetric signature algorithm. This crypto-agility [226] is especially important for future-proofing the protocol to remain relevant in a future post-quantum computing world.

SRUP messages themselves consist of a byte-stream composed of a concatenation of a number of fields. These fields consist of both the specific information to be communicated (e.g. instructing the device to perform a given command), and additional elements to ensure the integrity of the message. The hash digest of the message is signed using the sender's private key. Using this approach provides assurance that the message has not become corrupted or tampered with in transit, and provides a way to positively verify that the message has originated from the source that it claims to have come from. Although the protocol is independent of the specific cryptographic algorithms adopted, the reference implementation of SRUP (see Chapter 9), protects the messages by signing the SHA-256 hash value of the message with an RSA signature.

Regardless of the algorithm, provided that the device has a copy of the server's public key it can authenticate that any messages received genuinely originate, in the form received, from that server. Similarly any message from a device can be authenticated by the server, using the copy of the device's public key held by the server.

5.2 MQTT payload

Given the requirement for IoT devices to operate in constrained or austere network environments described in Section 1.2.3, it is important for SRUP to adopt an approach which maximizes efficiency when sending data. Therefore, whilst it would be perfectly possible to utilize a human-readable, ASCII text-based format such as JavaScript Object Notation (JSON) [227] for the elements of a SRUP message, this would require one byte per character, and therefore be very inefficient [228]. As such the format adopted by SRUP is to use a raw binary byte-stream, utilizing the position within the byte-stream to indicate the field-boundaries.

Although commonly used to send *printable* ASCII characters, the MQTT payload is defined as a simple byte-stream. As such, any raw data may be placed in the payload of a message that is published to the broker, and this data will then be sent to all subscribers.

As this raw binary format is supported by MQTT, all bytes are written directly into the MQTT payload (including elements such as the message signature). This is in contrast to the approach which would be required if using a text-based format, where constraining the payload to only ASCII characters would require the use of an encoding technique such as Base64 encoding [229] to represent the binary data using printable characters.

The elements contained within a SRUP message include both static and dynamic length fields. For the purposes of marshalling the data into a byte-stream, variable length fields are preceded in the byte-stream by a two-byte representation of their length; whereas fixed-length fields are simply appended onto the byte-stream. This means that a receiving system can parse the data very easily to reconstruct the message, and can allocate the correct number of bytes of memory for the variable length items, before populating the fields.

Although a number of data marshalling libraries (for example Google Protocol Buffers[1]) [230] are extant and in widespread use, they are relatively complex and heavyweight for the simple and fixed requirements of SRUP — and so a simple, bespoke, solution was adopted instead.

5.3 SRUP message elements

All SRUP messages are formed of a common set of fields, plus any message-type specific data that is required. This common base message, consists of the following elements:

- A two-byte header, consisting of:
  - One byte used to signal the version of the protocol in use.
  - One byte used to signal the message type.
- An eight-byte Sequence ID number: used to prevent replay attacks.
- An eight-byte Sender ID: this denotes the identity of the sender, and thus enables the receiver to ascertain the correct public key against which to validate the signature.
- A transaction token, used to identify a particular message thread, for multi-part operations. This consists of:
  - A two-byte token length.
  - A variable length token.
- The cryptographic signature, consisting of:
  - A two-byte length for the signature.
  - The signature data itself (of variable length).

The 8-byte (64-bit) sequence ID value is used to protect against a replay attack (see Section 8.1). It therefore requires that all devices receiving a SRUP message, must reject any messages where the sequence ID of the received message is not strictly greater than the last received sequence ID from that sender.

[1]https://developers.google.com/protocol-buffers/

5.4 MQTT topics and message addressing

SRUP messages do not contain an explicit destination field, but instead rely on the use of MQTT topics to route messages to the correct recipients.

5.4.1 Message source

SRUP uses the MQTT topic as an informational element within the messaging scheme. Each individual device uses a separate MQTT topic for its message traffic, and C2 servers subscribe to topics corresponding to devices that they control.

For example, consider a hypothetical example with the following three devices:

- `Device_0x01`
- `Device_0x02`
- `Device_0x03`

Each device uses an MQTT topic for its own messages.

- `SRUP/dev0x01`
- `SRUP/dev0x02`
- `SRUP/dev0x03`

(Note: in a real system the device ID would be a 64-bit randomly generated value, but for the clarity of this example, these rather more complex values have been substituted for more readable IDs).

If the C2 server wishes to send `Device_0x01` a message — that message will be published to the `SRUP/dev0x01` topic. A message from `Device_0x01` to the server will also be sent on the `SRUP/dev0x01` topic.

In a system where there is only ever one C2 server communicating with a device at any one time, no additional information is required to indicate the sender of a message. A C2 server will know which device a message pertains to because of the MQTT topic in use, and a a message sent to a device will always originate from that device's current C2 server.

Although this is the expected standard configuration for SRUP, there are scenarios where devices may have multiple C2 servers (for example where different elements of the device's operation are controlled by different C2 systems). In such a scenario where a device was receiving messages from more than one C2 server, this simple model is insufficient. In that scenario an additional mechanism is required to identify the sender, and therefore to identify which public key should be used to verify the message signature.

The simplest possible solution would be to have the receiver try each of the keys it has in turn before either verifying the sender, or concluding that the message cannot be verified. However this is neither a scalable nor elegant solution.

Another alternative would be to adopt a more complex MQTT topic hierarchy to signify the sender. For example, rather than subscribing to `SRUP/dev0x01` — the device and server(s) could subscribe to `SRUP/dev0x01/#` (using the MQTT topic wildcard character `#`). This would then enable messages from the device to use the higher-level topic — and for messages from different servers to use subtopics (e.g. `SRUP/dev0x01/sv0x2F`) to indicate their origination. The sender could then be identified by parsing the MQTT topic, and the subtopic element could then be extracted in order to determine the identity of the sender. This too was rejected for being unnecessarily complicated.

The third approach, which was adopted, is to include an additional field within the SRUP messages to contain the Sender ID. This method requires the smallest amount of work from the receiver to identify the sender, however it also has the largest additional overhead on the size of the message traffic. A typical SRUP base message has a length of 286 bytes. This is composed from: a 2 byte header, 18 bytes for the token (including its length), 8 bytes for the Sequence ID, 2 bytes for the signature length, and 256 bytes for the signature (assuming the Public-Key Cryptography Standard #1 (PKCS#1) [231] is adopted, as per the reference implementation, and that a 2048-bit key-length is used). With the addition of a 64-bit Sender ID field, the base message size increases to 294 bytes — approximately a 3% increase in size. Although not negligible this may be considered a preferable approach compared with the relative complexity of using the topic hierarchy-based approach.

As such, all SRUP messages have a sender ID associated with them, as a part of the base message. This ensures that (regardless of the C2 topology adopted within the implementation of any given system), a receiver of a message can always positively identify the sender and validate the message using that sender's public key.

5.4.2 Destination addressing

Using the approach described in Section 5.4.1, and by controlling the MQTT topics to which devices are able to subscribe, it may be assumed that any message received by a device is intended for that device. Although it would be possible to mirror the approach adopted for the Sender ID, and add a Destination ID field to the SRUP messages, an explicit destination ID was not included.

5.4.2.1 Positive device identification

The inclusion of a specific destination address would be consistent with lower-level networking protocols (such as TCP); however, unlike lower-level protocols this destination addressing is not required for routing (at either a highly abstracted logical level, or at the network transport layer level) — since the centralized MQTT broker and the publish / subscribe model take care of both of these. Furthermore because SRUP imposes a centralized C2 model, there is no scope for direct device-to-device messaging.

There is however, one type of SRUP message which does make use of an explicit destination ID: the software update initiate message. This is a deliberate exception. By adopting an explicit target identification field, it forces a strictly one-to-one mapping between specific devices and the corresponding update initiation messages. Although the topic addressing approach used elsewhere within SRUP can be used to sufficiently convey the destination, there is a requirement (articulated in conversations with potential users of the protocol) within some military and safety-critical applications to guarantee that device-specific payload delivery can be maintained. To provide such assurance the additional field was added to the protocol.

5.4.2.2 Multiple C2 servers

Given the all-to-all nature of the MQTT publish / subscribe model (where all messages published to a specified topic are relayed to all subscribers), a system with multiple C2 servers subscribing to a channel used for a specific device also have the problem of identifying to which of the potential recipients a message is addressed.

This is largely a problem which must be solved at the level of the design for a system using this model however, and is not a specific issue which SRUP is required to solve. There are a number of approaches that could be adopted by a C2 system to address the potential ambiguities of having multiple C2 reporting lines, which are well understood within the military domain. For example, a system could adopt segregation by message type (where messages of one type are handled by one C2 server, and messages of another type are handled by a different C2 node); a multi-level superior / subordinate hierarchical approach could be adopted (with some C2 operations delegated to the lower level, and others passed up to the superior commander); or control of a device could simply be handed over from one server to another, eliminating the situation where a device was simultaneously connected to multiple C2 servers. This latter approach, which is the expected mode of operation for the majority of situations where multiple C2 elements are required, can be implemented by simply ensuring that only a single C2 server is subscribed to a given device topic at any one time, and using SRUP join and remove message types (see Section 5.6.5) to explicitly handover control between C2 servers.

5.5 Message encryption and access control

If encrypted messaging is to be adopted within a SRUP system, then a mechanism is required to restrict access to the topics which can be subscribed to via the MQTT broker. Given the open device registration model (see Chapter 6), without such restrictions on topic subscriptions, anyone could simply register a device and subscribe to receive any messages: effectively bypassing the message encryption as they would become a valid recipient.

5.5.1 Encrypted messages and topic access control

When a C2 network is utilizing TLS for MQTT message encryption it is therefore necessary to both restrict access to the MQTT broker itself, and to limit the topics to which a given MQTT client is able to connect. In the context of SRUP, the broker needs specifically to be able to identify the client's identity, and then to use this to restrict the access for that client, so that it can only subscribe and publish to topics that are permitted for that identity, using the topic-based addressing scheme previously described.

This is accomplished by using a custom X.509 certificate to enable connection to the MQTT broker, and by configuring the broker to only permit connections from devices bearing a valid certificate. The certificates contain an MQTT username (stored within the CN field), and the certificate generation process ensures that the username matches the device ID. The broker then uses an Access Control List (ACL), which specifies that devices are only able to subscribe to a topic associated with their MQTT username. An example of this is shown in Figure 5.1.

In addition to having full read / write (*publish and subscribe*) access to their own topic, devices also require a means to send messages to servers that are not yet subscribing to that topic as a part of the process of joining a new C2 network. In order to accomplish this without breaking the model of restricting access to the topics, devices will initiate a request to join a C2 network, by using a topic associated with that particular server. For example, to join a network controlled by server `0xFEDCBA90`, the device would publish their join request message on topic `SRUP/SERVER/0xFEDCBA90`.

By using an additional level of topic hierarchy (`SRUP/SERVER/...`), together with an ACL written to also permit the use of wildcards, it is possible to permit any device able to connect to the broker to be granted *write-only* access to the topic of any server (e.g. to any subtopic of `SRUP/SERVER/`). The ACL rules for these *server topics*, will permit any client able to connect to the broker, to be able to **publish** to these topics, whilst limiting the ability to subscribe only to the servers.

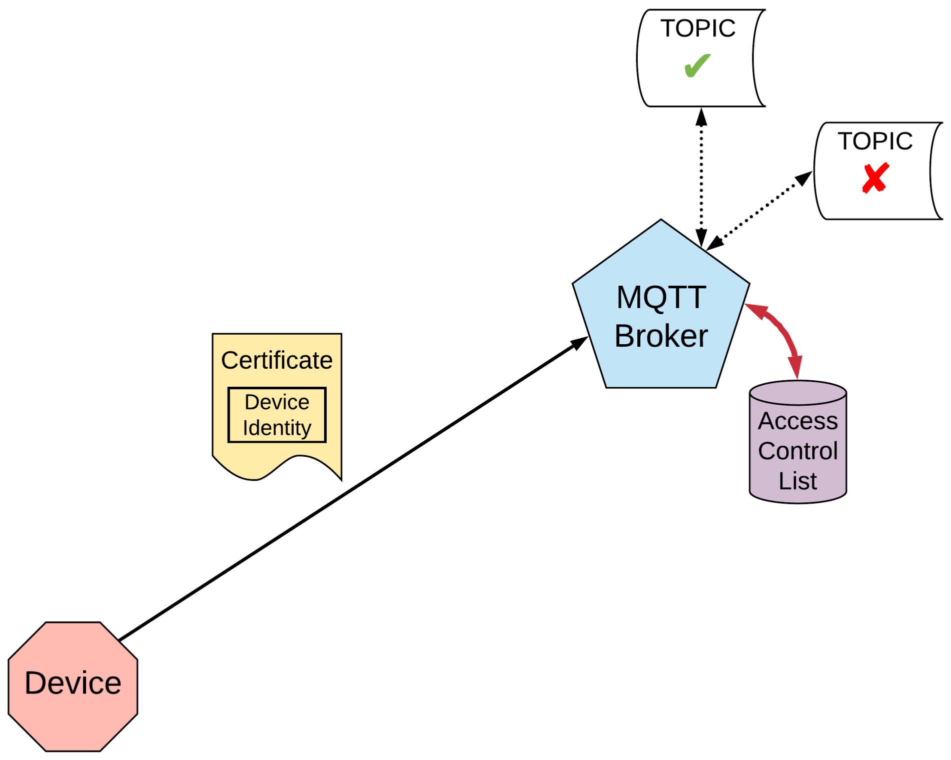


FIGURE 5.1: A diagram illustrating the use of *topic access control* to restrict access to MQTT topics by the broker.

For further details on how the MQTT broker ACL described here works with the identity model adopted within SRUP, see Chapter 6.

5.6 Message types

Within SRUP each message has a specific message type associated with it, signalling to the receiver which fields to expect to find within the message, in addition to the base message fields described in Section 5.3. As the message type is contained within the header, it permits a device receiving the message to read the header, to therefore know how to demarshall the data contained within the MQTT message.

These message types are grouped into families: each associated with a different type of C2 transaction.

There are seven main *families* of message type:

- Update Messages
- Response Messages
- Action Messages
- Data Messages
- Join Messages
- Remove Messages
- Syndication Messages

A summary of the first six of these families of messages, and their intended usage, is shown in Table 5.1. Note that the final version of the protocol also includes a number of additional message types to support the final message family, *syndication* messages. These are not discussed further in this Chapter, but full details of syndication, and the supporting message types are described in Chapter 10.

The following Sections contain a brief summary of the message types shown in Table 5.1; for full details of the messages, and detailed descriptions of the message fields contained within them, see the complete protocol specification in Appendix B.

5.6.1 Update messages

The most complex of the message families is the update message family. This consists of two discrete message types:

- Update Initiate Messages
- Update Activation Messages

The update process also makes use of the response message type (see 5.6.2) to signal the outcome of the update operations. An example of the update process using SRUP is shown in Section 5.7.

Message Type	ID	Family	Description
Initialize	0x01	Update	Used to initiate the software update process.
Response	0x02	Response	Used as a response message for all actions requiring confirmation.
Activate	0x03	Update	Used to instruct a device to switch to using the new software update previously received.
Action	0x04	Action	Used to command a device to perform an action.
Data	0x05	Data	Used to send arbitrary data between a server and a device, or *vice versa*.
ID Request	0x06	Action	A special case of an action message, requesting that the receiving device sends its identification string.
Join Request	0x09	Join	Used by a device to make a simple join request to a C2 server.
Join Command	0x0A	Join	Used by a C2 server to instruct a device to join a C2 network.
Human Moderated Join	0x0B	Join	Used by a device to request a *human moderated* join.
Human Join Response	0x0C	Join	Used by the C2 server to send the human join response code.
Observed Join Request	0x0D	Join	Used by a device to request a machine *observed* join.
Observed Join Response	0x0E	Join	Used by the C2 server to send the observed join response code.
Observation Request	0x0F	Join	Used by the C2 server to request that an *observer* performs an observation for a given join operation.
Resign Request	0x10	Remove	Used by a device to request to leave a C2 network.
Terminate Command	0x11	Remove	Used by a C2 server to instruct a device to leave a C2 network.
Deregister Request	0x12	Remove	Used by a device to inform a C2 server that it is permanently leaving a C2 network.
Deregister Command	0x13	Remove	Used by a C2 server to instruct a device that its identity has been permanently blocked within that SRUP *universe*.

TABLE 5.1: A table showing the main SRUP message types, and their intended uses

5.6.1.1 Update initiate message

The update initiate message is designed to initiate the process of causing a device to retrieve a software (or other configuration data file) update.

The message consists of the standard base elements, plus:

1. An eight-byte target ID — used to ensure that the received update message is intended for the receiving device.
2. A URL from which the software is to be retrieved, and a two-byte length for that URL.
3. A message digest for the software to be retrieved, and a two-byte digest length.

As described in Section 5.3, this makes use of a number of variable length fields. This ensures that elements such as the URL are sent as efficiently as possible (without padding) and without arbitrarily constraining the length of the URL that could be used. It is necessary to send two additional bytes for these variable length elements, denoting the length. Using two bytes ensures that there is effectively no practical limit on the length of the elements. $(2^{16} - 1) \equiv 65,535$ characters: more than long-enough for any currently conceivable URL or cryptosystem signature / hash value.

On reception of a valid message, the device should attempt to retrieve the data from the specified URL, check it against the hash value provided, and then signal back to the sender as to the outcome of this process — using a response message.

5.6.1.2 Update activate message

The update activate message is designed to trigger the receiving device to activate the software update which it has previously retrieved.

The use of an activate message enables the C2 server to stage the deployment of a software update. All devices can be signalled to retrieve the update using the initiate message, and then at a future time (after all of the devices in question have responded to inform the C2 server that they have successfully retrieved the update) all the devices can be activated *en masse*. This would be especially important if, for example, the software update for the devices was to switch them to use a new cryptographic key as part of their primary function.

The activate message contains no additional data elements, beyond those in the base message.

Further details of the security-related aspects of the SRUP update messages can be found in Chapter 8.

5.6.2 Response messages

The response message is used to signal the outcome of a variety of different SRUP operations. It consists of the base message — plus one additional element, a one-byte status. The values that may be taken by this status byte are defined in the specification (see Appendix B).

5.6.3 Action messages

The action message is a message type to be used when sending a request that the device performs some kind of action. The message consists of the base message — plus an additional one-byte Action ID. The meaning of any given value of this byte is user-defined, and therefore must be agreed upon for any devices and servers within a given system.

Receiving devices may optionally send a response message to signal the outcome of the request.

5.6.3.1 Identification request message

The identification request message is essentially a special type of action message, designed to provide a mechanism for the C2 server to request a state description or other identification string from a given node. Whilst the details of this are deliberately implementation specific, an example of this would be for a C2 server to request the version of the software running on that device, the device's serial number, or a hash of the exact version of the software that the device is running.

When a device receives a identification request message, it should respond by sending a data message containing an identification response string containing the expected data.

Whilst the generic identification request is included as a separate message type, in the event that a given implementation requires additional request types for identity or version information this may be accomplished by defining additional action message types for this purpose.

5.6.4 Data messages

The data message consists of the base message, plus two additional variable length elements:

- Data ID: an application specific value, used to denote the identity of the data being sent (and thus informing the receiver as to what to do with the data)
- Data: the actual data itself as an arbitrary-length byte-stream.

Note that both of these elements are variable length — and so are each delimited using two-byte length elements, also sent within the message type.

The details of the implementation of the data message are application specific — so, for example, the data type and semantic meaning of a value with a given data ID (e.g. `FAN_SPEED`) must be agreed between sending and receiving applications as a part of the implemented system specification.

In practice this means, that when the data message type is implemented, it is necessary for the receiving device to know what data type the received data should be interpreted as. This must always be explicitly interpreted by the receiving application, on the basis of the specified data ID, since the data contained within the SRUP message is just a byte-stream. So, in the previous example, both senders and receivers within a given system implementation must have an agreed definition that the data associated with data ID `FAN_SPEED`, will be (for example) an integer value.

5.6.5 Join and remove messages

The join and remove message families are used to negotiate devices joining or leaving the control of a specified C2 server. A number of different message types are specified in order to facilitate different types of join operation. In order to remove a device from the control of a server, SRUP provides resignation and termination message types. Devices may be permanently removed from the system via deregistration messages.

5.6.5.1 Join messages

The SRUP protocol identifies three types of join operation.

- Unmoderated, or simple, join
- Human-moderated join
- Machine-moderated, or observed, join

Unmoderated join operations are used in situations or systems where no additional confirmation of the device's identity is required. The C2 server simply accepts (or refuses) the request with no additional steps.

Human and machine moderated join operations are used for situations where it is necessary for the device to physically confirm that it has the same identity as the logical device making the request. See Chapter 7 for more information on moderated join operations.

Note that the protocol supports both device-led joining (where the device makes the request to join a C2 network), and also C2-led joining, where devices that are already a part of one C2 network may be given a command to join a different C2 network. This second C2 network must be a part of the same SRUP *backend* system (also referred to as a SRUP *universe*). See Chapter 6 for more information on the SRUP backend system.

5.6.5.2 Remove messages

Similarly to the join messages — SRUP also supports removal of devices from a C2 network via distinct message types. Devices may request that they are removed from the C2 network, or the C2 server may issue a termination command to inform a device that it has been removed from a C2 network.

5.6.5.3 Deregistration

Distinct from removal messages, deregistration messages cause the device's public key to be deleted from the system. Devices should also remove the server public keys that they hold, upon notification of deregistration.

As with removal, the deregistration operation can be initiated from either the device (as a request) — or the C2 server as a command. Similarly, since only the device's C2 server may send a message to a device to force it to deregister, there is no scope for abusing this message type as a part of a *deauthing* attack.

5.6.6 Syndication messages

Although not originally a part of the SRUP protocol, before the completion of the research work, an additional family of message-types was added to support *Syndication* operations. Please see Chapter 10 for further details of these message types.

5.7 SRUP in action

In order to better understand how the protocol works, an example of information flow between a C2 server and a device during a software update is illustrated as a sequence diagram, in Figure 5.2.

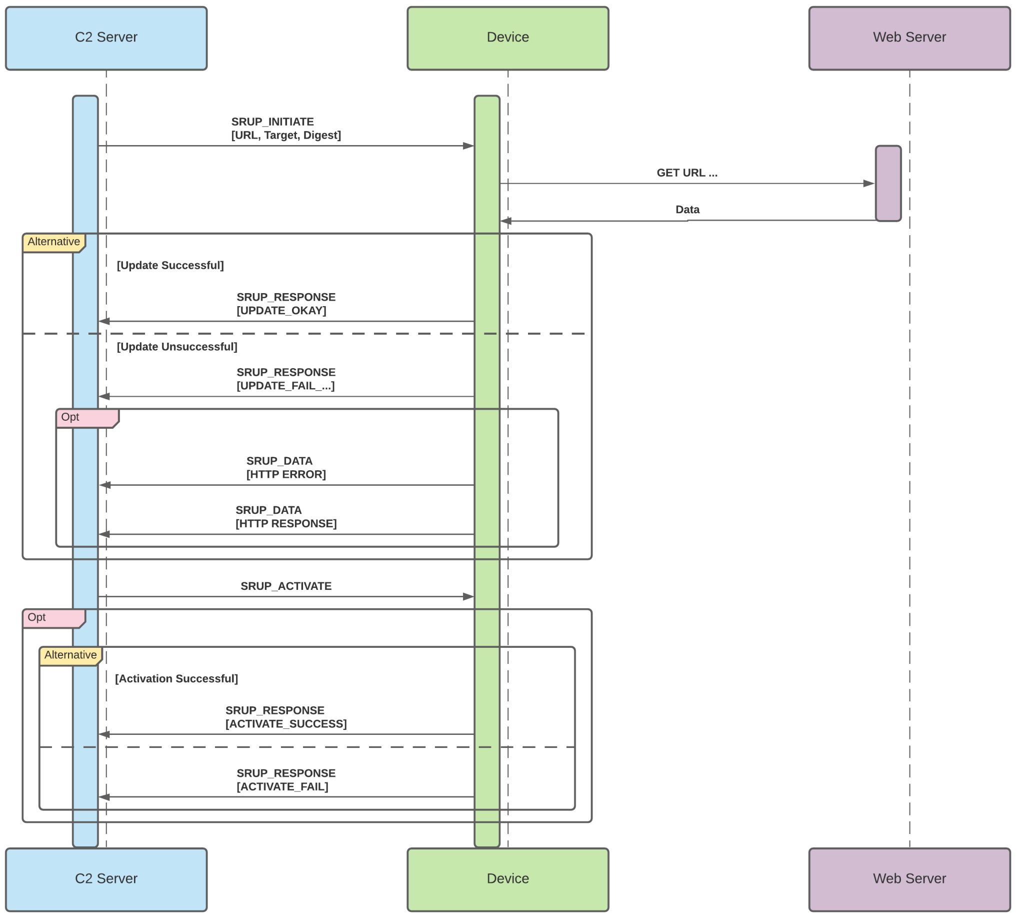


FIGURE 5.2: A sequence diagram illustrating the data flow during a SRUP software update operation.

The server is requesting that the device in question retrieves an update data file from a specified location, and that it applies the update according to the (implementation specific) update process.

To do this, a SRUP `UPDATE_INITIATE` message is first created.

The message fields corresponding to this message are shown in Table 5.2, and result in the byte-stream shown in Table 5.3. This byte-stream would then form the payload of the MQTT message.

Note that to aid exposition — the binary SRUP fields (such as the signature data) used here all take illustrative example values.

Element	**Value**	**Length**
Version	`0x03`	1
Message Type	`0x01`	1
Sequence ID	`0xB0308F9C43E22A3B`	8
Sender ID	`0x6E4DA844B26DA064`	8
Token	TOKEN	5
Signature	SIG_DATA	8
Target	TARGET	6
URL	https://www.example.com	23
Digest	DIGEST	6

TABLE 5.2: The elements of an example SRUP update initiate message

Hex Data	**Meaning**
`03`	Version
`01`	Message Type
`B0 30 8F 9C 43 E2 2A 3B`	Sequence ID
`6E 4D A8 44 B2 6D A0 64`	Sender ID
`00 05`	Length of Token (5)
`54 4F 4B 45 4E`	ASCII String: `TOKEN`
`00 08`	Length of Signature (8)
`53 49 47 5F 44 41 54 41`	ASCII String: `SIG_DATA`
`00 06`	Length of Target (6)
`54 41 52 47 45 54`	ASCII String: `TARGET`
`00 17`	Length of URL (23)
`68 74 74 70 73 3A 2F 2F 77 77 77 2E 65 78 61 6D 70 6C 65 2E 63 6F 6D`	ASCII String: `https://www.example.com`
`00 06`	Length of Digest (6)
`44 49 47 45 53 54`	ASCII String: `DIGEST`

TABLE 5.3: An expansion of the raw bytes of a SRUP update initiate message

These 76-bytes form the payload of the MQTT message which is sent to the broker, and then on to the subscribing device.

Having received the message, validated the signature, and checked the sequence ID, the device would then attempt to retrieve the data file containing the update from the specified address via HTTPS. If the specified end-point requires authentication, then it is assumed that the device will already have whatever access credentials are required. These may either be hard-coded into the device's current software, or may have been sent previously in the form of a data message.

If the data has been successfully downloaded to the device, the SHA-256 hash value for the data file will be calculated. This hash will then be compared to the value specified in the SRUP message. The hash values must match for the device to send a *successful* status within a response message.

Alternatively, if the data cannot be received (either because the web-server returned an HTTP 404 code to denote that the specified file did not exist; or because the server itself did not respond, such as in the case of the server being off-line); or if the hash did not match, then this information will be used as the status of a unsuccessful SRUP response message.

The token specified in this response message must match the one sent by the server in the update initiate message to enable the server to correctly associate the received response message with this update request.

The final part of the process is for the control server to send a SRUP message to the device to tell it to activate the newly received update. Again, the token sent here must match the token used elsewhere in this update operation. The server should only send the activate message after receiving a response signalling an update success. A device receiving an activate message associated with an unknown token, or a token for which no update success message was sent, should ignore the activate message.

5.8 Alternative transports for the SRUP protocol

5.8.1 The need for alternative transport mechanisms for SRUP

MQTT is an ideal protocol for applications where network connectivity is intended to be continuous (even if unreliable). Using appropriate MQTT QoS settings, will enable a sender to be confident that a target device will receive a message providing that device remains connected to the broker. The MQTT standard usually requires that at least one message is sent by a device within 1.5 times the specified *keep-alive time*. This message may either be a normal MQTT message or an MQTT `PINGREQ` message (which is used to signal that the device is still connected — but that it has no data to send). The MQTT standard defines the maximum keep-alive time as a 16-bit value expressed in seconds, and hence the maximum permitted value is a little over eighteen hours, as shown in Equation 5.1.

$$2^{16} - 1 \text{ seconds} \equiv 65535 \text{ seconds} \equiv 18 \text{ hours, } 12 \text{ minutes \& } 15 \text{ seconds} \tag{5.1}$$

This means that the network connectivity of a device must (effectively) guarantee that at least one MQTT message or `PINGREQ` can be sent by the device within a time period of no more than 27 hours (18×1.5), or more quickly, if a shorter keep alive time is defined by the broker configuration.

However for some applications, especially remotely deployed sensors intended to run for extended periods of time, a device may be totally dormant and disconnected from the network for periods far exceeding this maximum keep-alive time. Although in some brokers keep alive can be disabled by specifying a zero value, MQTT is not well suited to this type of use requiring extended open-ended periods of disconnection.

Devices may operate on this type of schedule for a variety of reasons — such as for operational emissions control purposes (such as devices wishing to remain covert, without signaling their position or existence), or for power consumption reasons (such as wishing to maximize the life of the battery).

For such devices, instead of waking every 27-hours, it may be required that the device may be awoken only once per week to check for messages. Moreover in many scenarios this interval may not be predictable. Periods of dormancy may be triggered externally (by the device itself, or the user of the device) due to the specific operating circumstances or environmental conditions at hand. In such a situation it is highly likely that the device would be required to remain dormant until such a time that a second physical action triggers the resumption of connectivity.

Although this use-case is somewhat outside the typical use-case for IoT devices, other types of remote sensing devices, or CPS may have requirements of this type, and these may also have security and authenticity requirements that could be satisfied by the use of the SRUP protocol.

Open-ended long-duration schedules such as these, would be incompatible with MQTT, but they could be achieved by using SRUP with an alternative application transport layer.

5.8.2 HTTP transport for SRUP

Although the SRUP protocol is designed to work using MQTT as the *application transport mechanism*, the protocol itself is abstract from that underlying mechanism. This permits the substitution of the transport for any given system or, by the use of a protocol bridge, the use of an additional alternative transport for some parts of a system.

For example, the device could connect to an HTTPS server (at a predetermined address) and retrieve data. This could be a simple JSON file (with each element of the SRUP message to be sent or retrieved, provided as a discrete JSON field), or to minimize the data file size an equivalent binary serialization format such as MessagePack [232] could be used.

Since the SRUP specification defines the order of the fields, the additional overhead of identifying the fields by name is not strictly required. For maximum efficiency a pure binary data format similar to the *on-the-wire* structure of the MQTT payload in *MQTT SRUP* could also be used.

Regardless of the data format involved, the device would be allocated a unique URL endpoint (using the web-server, effectively, as a communications broker), to which the device would issue a `GET` request.

For a further reduction in network bandwidth — the HTTP server could be substituted for a CoAP server, although in this context the client and server would be reversed from the usual CoAP architecture (where a device would typically be the CoAP server).

Multiple messages may be sent from a C2 server to the device during the time that the device is dormant, so it will be necessary that the platform acting as the message broker be able to queue multiple messages for a given device.

The simplest way to do this would be to encapsulate multiple messages into a JSON structure, where each message is a discrete object within the larger JSON data file. The downside of this approach is that the device may be required to receive a large block of data upon connection, in the event that a large number of messages had been sent during its dormant period.

A preferred approach may therefore be to have the web API endpoint return data (such as a JSON object) containing the total number of messages to be retrieved. The device could then retrieve these individually or in bulk.

For example:

```
GET http://server/device12345/

     {"message_count": 25}

GET http://server/device12345/message/7

     {"message\_ID": 7, {<SRUP MESSAGE>}}
```

Finally this approach could be combined with a prioritization system which could be used to differentiate between urgent and non-urgent messages...

```
GET http://server/device12345/
```

```
        {"urgent_message_count": 14, "other_message_count": 11}

    GET http://server/device12345/urgent/3

        {"message_ID": 11, {<SRUP MESSAGE>}}
```

Clearly this can be extended to a system which uses a rule-set to rank messages for delivery, such that if a device is only able to connect for a very-short period it can ensure that the most critical message(s) are retrieved ahead of other messages.

Similarly an HTTPS transport can be used to support messages from the device to the server. In this case the device would send a `POST` request to the web-application server end-point which would send a JSON (or other format) data message to the server. Obviously a similar approach to message queuing could be adopted for messages originated from the device during a period of device network dormancy, subject to storage constraints of the device.

Other transport mechanisms could also be adopted. However, the two-way nature of the C2 information flow requires a mechanism that supports both uplink from, and downlink to, a target device. This could include a low-power, long-range protocol such as LoRaWAN [233]. Consideration would need to be given as to the suitability of any particular implementation based on the requirements of the frequency and size of messages, and as to the likely proportion of messages originating at the devices versus those originating at the C2 server.

5.9 Summary

This Chapter has examined the design concept for the Secure Remote Update Protocol, the types of message that the protocol supports; and considered how the protocol can be used with an MQTT publish / subscribe architecture. The next Chapter will explore questions around identity management for IoT devices, and an automated identity management scheme to be used in conjunction with the SRUP protocol.

Chapter 6

Identity and Key Distribution

This chapter examines the challenges associated with the problem of identity management, and key distribution within the IoT, and its application within SRUP. It will examine two methods of identity assignment for IoT devices, considering the advantages and disadvantages of each. The material in this Chapter addresses Research Question RQ2, and represents the second original element of the research.

Elements of this Chapter have been previously published as [3].

6.1 Identity and the Internet of Things

The Secure Remote Update Protocol described in Chapter 5 makes extensive use of device identity. In this context *identity* consists of a label (unique within a given system) which provides an identifiable name for individual devices or servers. If identities are assigned at random (which reduces the likelihood of an attacker guessing a valid identity), then in order to guarantee uniqueness the pool from which identities are assigned needs to be sufficiently large, coupled with a guarantee of uniqueness of generation such that there is no possibility of collision of identity within the system. An easy way to achieve these requirements is to make use of a Universally Unique Identifier (UUID) [234]: such as may be generated using the Linux `uuidgen` command, or Boost C++ libraries [235]. UUIDs are 128-bits in length, and when using the randomly generated UUID version 4 variant, the number of random version-4 UUIDs which need to be generated in order to have an approximately 50% probability of at least one collision (as given by solution of the *birthday problem* [236]) is shown in Equation 6.1.

$$n \approx \frac{1}{2}\sqrt{\frac{1}{4} + 2 \times \ln(2) \times 2^{122}} \approx 2.71 \times 10^{18} \tag{6.1}$$

This number is equivalent to generating 1 billion UUIDs per second for about 85 years, which is more than enough to ever avoid collisions in any conceivable system.

In a system where cryptographic keys are used to assure the authenticity of messages, it can be asserted that *having a given identity*, has equivalence with the device in question *having a copy of the private key corresponding to the public key associated with that identity's UUID*.

Depending on the specific implementation to be used, the device identity could be assigned by the server (thus guaranteeing uniqueness for a given system), or it could be generated by the device itself. In this second case, it is still necessary for the server to validate the generated identity, in order to ensure uniqueness within the system. Although a randomly generated 128-bit UUID is extremely unlikely to ever result in collision (hence the fact that such values are used as an Universally Unique Identifier), the validation is necessary to prevent a deliberate collision attack against the system.

There are two approaches to defining the identity of a hardware device. To either use a *static* device identity, or a *dynamic* identity.

6.1.1 Static device identity

The first of these approaches is to adopt a static identity. In this model, each device is permanently issued with a fixed (and unchangeable) identity at, for example, the time of manufacture. This permanent assignment may be carried out by *burning* the identity value into ROM, Programmable Read Only Memory (PROM), or any other write-once media type. The storage media used for this either needs to be an integral part of the device (to prevent its physical replacement by a new storage device, containing a new identity value), or to be digitally signed by a suitable Certificate Authority to ensure it cannot be undetectably modified. A related approach is the use of a cryptographic storage device [237], such as a Trusted Platform Module (TPM) [238] to store the identity value.

An alternative approach for the issuance of a static identity, is to derive the identity using intrinsic physical characteristics of the hardware — such as Physical Unclonable Functions (PUFs) [239].

6.1.2 Dynamic device identity

The alternative method, which alleviates the need to rely on predetermined fixed identities, is to use dynamically generated identities. This technique creates a unique identity only at the time that a device is *registered* with the system, and allows that identity to be revoked and subsequently a new identity may be created.

Adopting this approach has a number of advantages. It eliminates the requirement to securely generate and distribute the keys associated with a fixed identity conferred at the time of manufacture. Given that it also enables the identity associated with a device to be revoked, and for a new identity to be assigned, this approach makes it much easier to ascertain that any previous access to a device has been revoked since the identity to which any previous permissions applied no longer exists. This concept is especially important for high-value Internet-connected devices, such as cars, which may be expected to have more than one owner during their lifetime. When utilising a fixed identity it is not possible for the new owner of a device to be certain that all previous owners have relinquished all access to the device; but when dynamic identities are used, the new owner may simply generate a new identity for the device and delete the old identity, guaranteeing that no other user has the security credentials for the newly created identity.

Dynamic identity also has the advantage that a device may be joined to any compatible system—with the minimum prior knowledge of that system. Providing the device knows the URI end-point address for the system's key registration service (and the core protocols that the system uses), all other data can be bootstrapped. Moreover, the C2 service does not require any previous relationship with the device manufacturer or originator.

6.1.3 Registering a dynamic identity

When initially registering with the system (or when replacing a previous identity with a newly created one), devices are able to establish contact with a web application over HTTPS to request an identity (and to perform the subsequent key exchange with the system). By using HTTPS, the device is able to positively determine the identity of the web service, and prevent a *man-in-the-middle* attack versus the key exchange process. This may exploit public CAs for Internet-based resources (where an appropriate root CA certificate is already present within the root of trust on the device), or by the *a priori* provision of a private CA certificate to the devices for systems designed to operate on private networks. Using HTTPS also ensures that the traffic between the device and web application is encrypted against eavesdropping.

6.1.4 Using dynamic identity in a C2 system

Once a device has an established identity, that device may elect to join a C2 network, either autonomously or as a result of a user-interaction. In low-security scenarios a C2 server may be configured to permit joining of any devices without further establishment of their credentials and as such, a *simple join* operation may be conducted.

Since any device joining the system may have just generated (or regenerated) its identity it is not possible for the C2 server (or a human operator of that C2 server) to ascertain the physical device to which that identity pertains. For low security systems (those where there are no sensitivities to the data and where the system itself is not controlling critical operations), this may be regarded as acceptable. However, for systems where there is the risk of an attacker injecting false data into the system, such an approach cannot be adopted. Solutions addressing this requirement are discussed further in Chapter 7.

The behaviour governing what types of join can be permitted (either globally, or on a device type basis) is determined in software by the C2 system.

6.2 Cryptographic key distribution and the Secure Remote Update Protocol

If a message is to be signed using asymmetric cryptography then the sender will require a copy of a suitable private key, and the receiver will require a copy of the public key associated with that private key. If it is to be encrypted, then the sender needs access to the public key that is associated with a private key held by the recipient. In both scenarios it is essential that the private key is not made available to any party other than the authorized holder. Therefore a safe, secure, and convenient mechanism for public key distribution is essential to ensure the integrity of secure messaging.

Consider an example IoT system using the SRUP protocol introduced in Chapter 5. This example system consists of an Internet-hosted C2 server, and one or more devices under the server's control. In order to perform the initial registration of the device with the system, there are five pieces of information that need to be exchanged between the server and the device.

1. The device needs to know the address of the MQTT Broker being used to route messages
2. The device needs to know the identity of the server to which it wishes to become subordinate
3. The device needs to know the server's public key, in order to validate messages from the server
4. The server needs to know the device's identity
5. The server needs to know the public key corresponding to the device's identity

6.2.1 SRUP key exchange via an HTTPS secure web service

The mechanism adopted to conduct the initial registration process within SRUP is to provide the device with an HTTPS URL which addresses a website to be used to securely exchange the data. By using a TLS protected connection to the web server the data is encrypted in transit, and since TLS also provides proof of the identity of the web-server the device is able to trust that the exchange process is taking place with the correct party. By using such a mechanism, SRUP does not need to solve the key exchange problem within the protocol itself. Providing the CA used to sign the site's certificate is within the root of trust for the device, then TLS solves the problem of key exchange for the HTTPS connection by using the standard web-based key signing X.509 certificates. Since HTTPS over TLS provides this, it is not necessary to re-implement the process within SRUP.

Therefore to use this approach for the distribution of the server's key, all that is required is that a C2 server (or a suitable proxy within the wider system) has a suitable certificate and REST API, and to arrange that this will provide the data that the device requires.

Sethi, Arkko, Keranen, *et al.* independently describe a broadly similar approach (albeit based around the CoAP protocol) for very simple devices, although their approach does not utilize the web of trust available by using public root CAs described here. Instead the approach adopted was for their devices to generate their own identities and for these to be communicated *"... out-of-band to the peers that need to know what devices to trust ..."* [240].

6.2.2 SRUP registration and key exchange workflow

Using the approach described in the preceding Section, SRUP requires a suitable key-pair to be used by both the device and the C2 server when communicating via SRUP messages. Additionally, for systems electing to use MQTT over TLS for message security, the service also issues the device with a TLS certificate and key, enabling the device to participate in encrypted communications with the MQTT broker (as described in Section 5.5).

The combination of the SRUP backend (the key exchange service and secured MQTT broker) and the underlying cryptographic system being used for that specific instance of SRUP traffic can be referred to as a SRUP *universe*. Syndication operations involving communications between C2 systems in multiple SRUP universes are discussed in Chapter 10.

The specific process for the initial registration (up to and including the operation to join the relevant C2 network) generically consists of the following 10-step process. For a system not using a TLS encrypted MQTT connection, steps 6 – 7 are skipped.

1. A new device connects to the initial registration server URL to retrieve the status, and ID of the server.

2. The device then sends a POST request consisting of the UUID device ID, a locally generated public key for the device, and any additional information that the specific implementation may require such as a device type identifier. This data is enclosed within a JSON object communicated using the REST API.
3. The server responds by sending the MQTT broker URL, the identity of the *default* C2 server for that given universe, and a copy of that server's public key. Again, this is sent as a JSON object.
4. The device then signs its identity with its private key and sends this to the server.
5. The server is now able to validate the received public key for the device — and reciprocate the process.
6. The device having received the server's signed identity and validated the previously received server public key, may then generate a second key for use with TLS protected MQTT, along with a CSR with the CN field set to equal the device's identity. This is then sent to the server, together with a signature generated using the C2 protocol key.
7. If the data received by the server is valid, and the CN field is equal to the identity possessing the key used to sign the message data — the server generates a device certificate, signing it with a SRUP CA root key. The server then sends the device certificate and the CA certificate to the broker.
8. The device now connects to the MQTT Broker specified in step 4 — using the device MQTT key and certificate (and the CA certificate), if required.
9. The broker will permit the device to subscribe to the topic associated with its unique identity.
10. The device may now issue a join request to join the C2 network.

6.2.2.1 Initial registration

The initial registration process corresponds to steps 1–3 of the 10-step process described in 6.2.2, and is illustrated as a sequence diagram in Figure 6.1.

The registration process can take place without any human interaction. The device connects to the registration URL, and after communication has been established, it sends a POST request to send its identity, its public key, and any additional information about the device that the specific system implementation requires. In return, the response from the server will contain the URL for the MQTT broker, and the default server's public key. The mechanism to store the server key and URL on the device will naturally be device-specific, but could consist of the device's file-system or a dedicated flash memory or Electrically

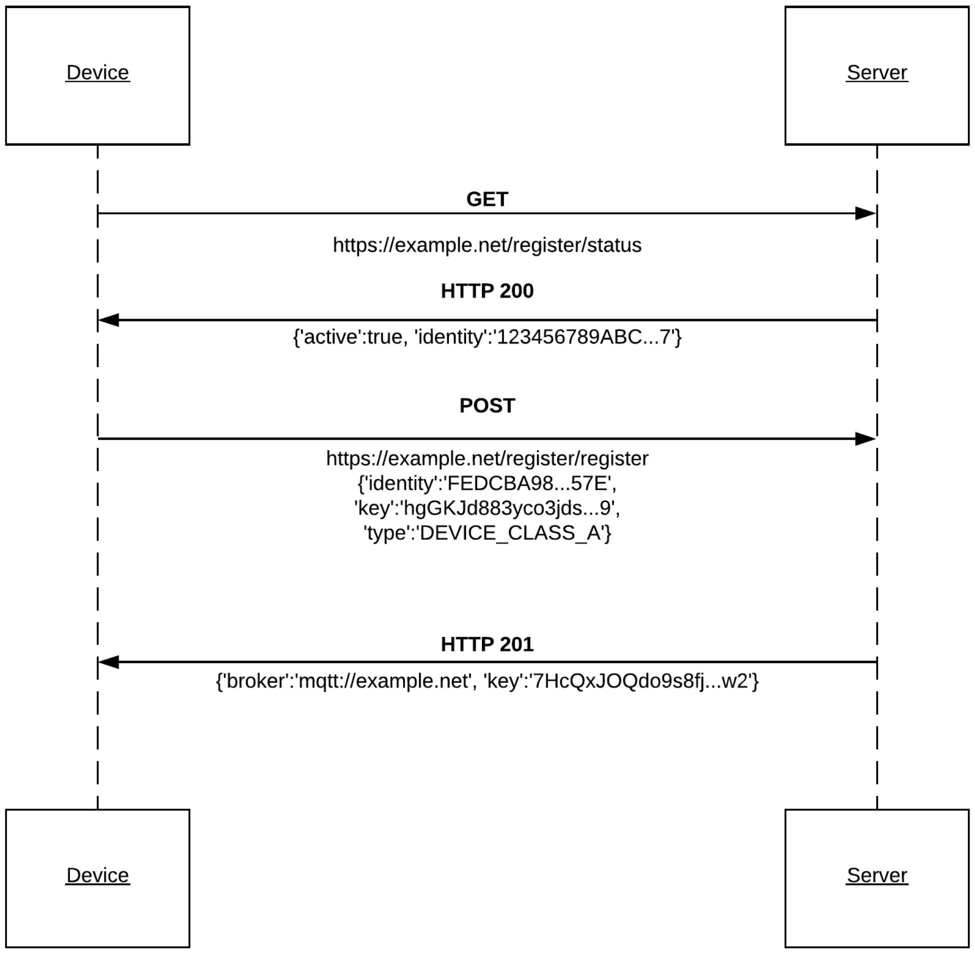

Device
Server
GET
https://example.net/register/status
HTTP 200
{'active':true, 'identity':'123456789ABC...7'}
POST
https://example.net/register/register
{'identity':'FEDCBA98...57E',
'key':'hgGKJd883yco3jds...9',
'type':'DEVICE_CLASS_A'}
HTTP 201
{'broker':'mqtt://example.net', 'key':'7HcQxJOQdo9s8fj...w2'}
Device
Server

FIGURE 6.1: A sequence diagram illustrating the key exchange and registration process for a new device (steps 1–3)

Erasable Programmable Read Only Memory (EEPROM) module. On the server-side it would be expected that this information would be stored on a suitably protected private database.

Registration requests can be received from any device without prior arrangement. Simply having registered a device on the system does not establish any specific relationship between that device and any server, nor does it assign trust to the device. As such, an attacker registering one or more devices with the C2 server has very little impact on the operation of the system.

Given there is no requirement for manual intervention within the initial registration process — it can be considered to scale very well, even to extremely large numbers of devices. The only constraints on the number of devices that can register this way are the storage on the server issuing the keys, and the bandwidth / availability of the server to receive multiple simultaneous connections. Both of these can be addressed with the provision of scalable cloud services — such as Amazon Web Services (AWS) Elastic Cloud Compute (EC2) [241].

In the edge case of a system that may be (or is being) subjected to a DoS attack, where an attacker attempts to overwhelm the registration web-server by sending an extremely large number of mendacious device requests, some mitigation may be provided by filtering or otherwise restricting registration requests. The utilization of modern good-practice implementation of web services, to provide load-balancing to the server backend (both at the web-service layer, and also the underlying database mechanism used to store the identities), may also go some way to help to mitigate this type of attack. In order to prevent a DoS attack on the registration service causing disruption to the operation of devices already using the C2 system, the use of a microservices-based architecture should be adopted to prevent the MQTT broker and C2 system's operation from being affected.

Filtering registration requests through the addition of HTTPS authentication could be trivially added to restrict access to the registration end-point. This could be on an individual device basis, or (for devices permitting richer human-computer interaction) on an individual registrant basis (see Section 6.2.3).

6.2.2.2 Additional steps for systems using TLS protected MQTT

For systems which are to use a TLS encrypted MQTT connection — an additional set of steps (corresponding to steps 6–7 in the list shown in Section 6.2.2) are required.

The device making the request must first generate a new key to be used solely for the MQTT over TLS connection. It must also generate a CSR for this key. The CSR's CN field must be set to be the device's identity. This CSR is then sent to the server, along with the identity of the device, and a signature derived from the identity and CSR data. In keeping with the common conventions for REST API end-points, all of this data would be sent as a JSON object, using base64 encoding [229].

On receiving this, the server must check that the signature is valid — and that the CN field of the CSR matches the device identity of the device possessing the key used to sign the message. Assuming that it does, the server will then generate a certificate (using the system's CA key), which can be used along with the public-part of the TLS key when connecting to the MQTT server. Since the CA key is used only for private communications with the MQTT broker there is no requirement for the CA key to be signed by a publicly known root CA certificate. The broker may use a signed CA, or it may generate its own *self-signed* CA key. However, since the device will not recognize authority of a self-signed CA key — the server must also supply a copy of the CA certificate to the device, when it returns the newly created device certificate.

A sequence diagram outlining these steps can be seen in Figure 6.2.

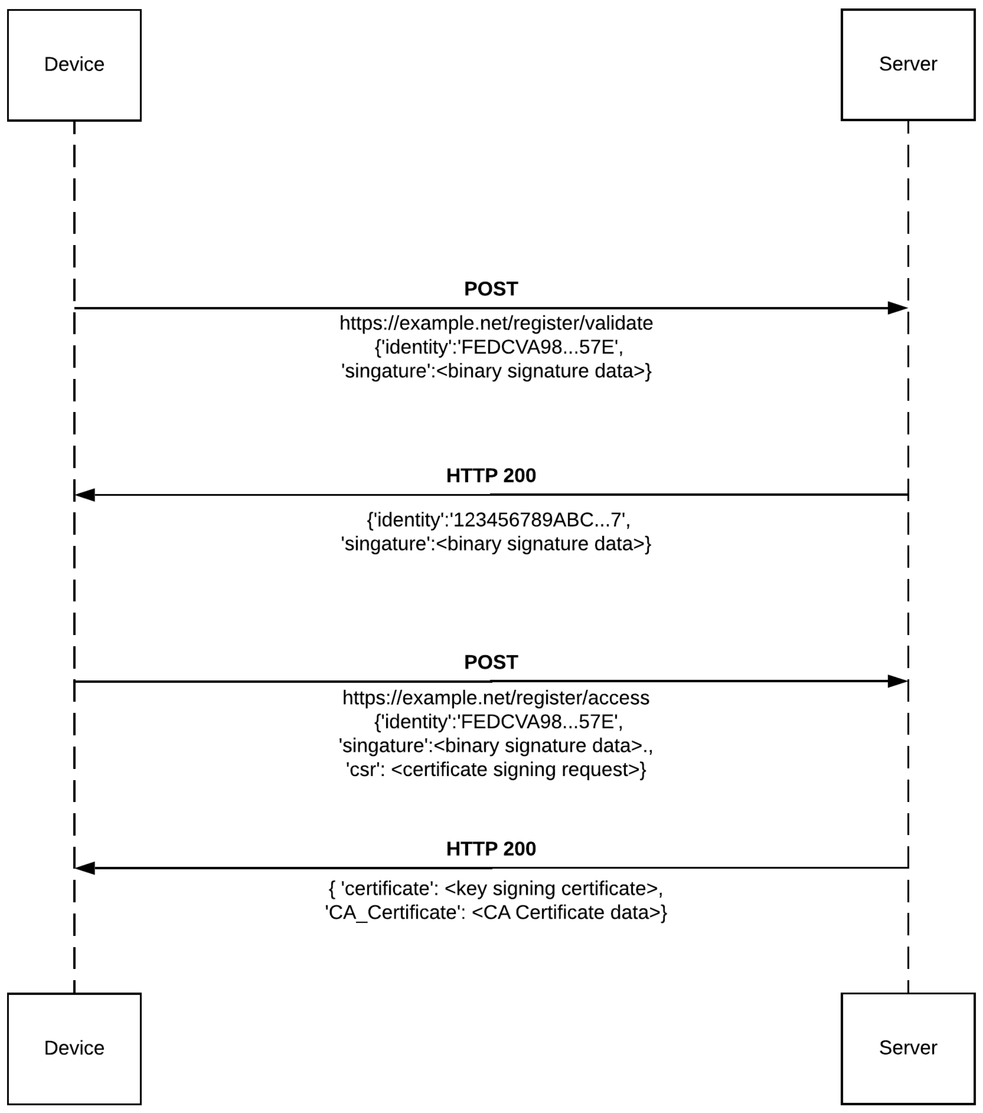

Device
Server
POST
https://example.net/register/validate
{'identity':'FEDCVA98...57E',
'singature':<binary signature data>}
HTTP 200
{'identity':'123456789ABC...7',
'singature':<binary signature data>}
POST
https://example.net/register/access
{'identity':'FEDCVA98...57E',
'singature':<binary signature data>.,
'csr': <certificate signing request>}
HTTP 200
{ 'certificate': <key signing certificate>,
'CA_Certificate': <CA Certificate data>}
Device
Server

FIGURE 6.2: A sequence diagram illustrating the additional key exchange process for systems using MQTT encryption (steps 4–7)

6.2.3 Communicating the registration URL

Providing that there is physical access to the device at the time it is deployed, communicating the initial registration URL to the device is very straightforward. It could be hard-coded into the device, or the user deploying it could manually enter the URL in full (or using a public or private URL shortening service — assuming that such a service is considered suitably trustworthy). A non-textual mechanism for encoding of the URL could also be adopted. For example, this could be in the form of a QR-code [242] or other two-dimensional bar-code, or it could use radio-frequency communications via an Near-Field Communication (NFC) tag [243] or a Bluetooth Low Energy (BLE) beacon [244] protocol such as Eddystone [245]. If using this approach, combined with a device which supports an HCI permitting it, the human registrant could also use individual registrant credentials for the registration process.

For a system where it is not possible or desirable to have physical access to the device at the time of registration it would be necessary to adopt a hard-coded approach of providing this initial registration URL in the device's software or firmware. This is clearly less flexible as an overall solution since it constrains the device to only be usable in the context of one specific system, however for many systems this may be considered a desirable feature, as it would lock the device into only being used as a part of the system in question.

Whatever the means of providing the URL, it is to be expected that in the majority of cases the registration process will take place before the device has been deployed. This means that despite the potentially limited bandwidth available to the device in the field, the registration operation can be assumed to occur without specific bandwidth constraints, and hence the additional overheads of a HTTPS connection are not regarded as an issue.

6.3 Key revocation

Key revocation is a vital, but often overlooked [246] aspect of any IoT system. It is, however, simple to implement in the context of a C2 based approach to an IoT system utilizing dynamic identities.

There are two types of revocation that need to be considered.

1. The removal of a device as a subordinate to a C2 server such that it is no longer subject to control by that server whilst retaining the device within the wider system. This is the opposite of a join operation.
2. The permanent withdrawal of the device from the system as a whole. This is the opposite of initial device registration.

These are associated with the Remove and Registration message types described previously (see Section 5.6.5).

To remove a device from a particular C2 network, the device may either elect to resign by sending a removal request message, or it may be removed by the server by being sent a removal command message. Given the hierarchical nature of the communications — any device is expected to comply with such a command. However even if it does not (such as may be expected from a malicious or otherwise compromised device), the C2 server will have unsubscribed to the MQTT topic pertaining to the device — and as such, the device will not be able to communicate with the system, even if it is still attempting to do so.

If the device is being disconnected by the server, the server would simply remove the record pertaining to the device from its list of subordinate devices. If the device is requesting to resign then the server should send a response message with a status to indicate whether or not that resignation is accepted. If it is, then the key would be removed as previously described.

For a permanent, and system-wide deregistration of a device, a C2 protocol can also be used — despite the fact that the initial registration took place outside of the protocol. The data exchange is essentially the same as for the resign / terminate use-cases, but with specific message types associated with removal from the system with the device's key being permanently removed from the system's key store.

The permanent deregistration of a device is intended to be used in situations where the device in question is being withdrawn from the system. For example, an environmental monitoring device may be registered before being deployed to gather data. On the completion of its deployment it may then be withdrawn, and deregistered to prevent it accidentally being rejoined to a C2 network. The device may be redeployed at a future time, by simply re-registering it and then using that new identity to join the new C2 network.

Deregistration can also be used to revoke access for a device known, or suspected, to be compromised (physically or logically) by an attacker.

Within the SRUP protocol, identity revocation can be conducted either from the device (for example as a result of user interaction), or from the C2 server. Revocation from the device ensures that the old identity no longer exists, regardless of whether it remains registered within the C2 system. This has the advantage that it does not require consent from the C2 system, and enables the device to join a new (or rejoin the previous) C2 system under a new identity. Devices should send a *deregister request* to inform the server that their identity is being revoked, but this is not mandatory. Revocation from the C2 server will result in the device being sent a *deregister command* message. On receipt of this the device should revert to an *unregistered* state since it will no longer be able to communicate with servers using the identity it currently holds. Within the protocol it is not possible for a third-party to cause the revocation of an identity, which would otherwise open an attack surface for attacks against devices.

6.4 Server configuration and identity

The process for device identity management is intended to be automatic, so that devices can register with the service automatically as required. However, the process for registering a server with the key service was deliberately designed to require intervention from a human administrator. In order to provide an additional validation step, so only valid C2 systems can be registered with a given SRUP backend, it is necessary to generate a *server token file*. This file consists of a base-64 encoded [247] version of the RSA signature generated when the future server's public key is signed by the backend system's (key exchange server) private key. When a server attempts to register with the key service, it is required to send this data, along with its public key. The backend system can then validate that the request is being made by someone who has (or who has had) administrative access to the backend system within that universe. This additional generation step can be easily accomplished by an administrator with access to the backend system, but ensures that only manually approved servers can operate within that *universe*.

6.5 Summary

This Chapter has introduced the concept of dynamic identity, and has examined how this can be used within the context of the IoT and SRUP — a topic which will be explored further in the next Chapter. The key distribution model used within SRUP has also been presented, and the required information exchange process has been explored. In the next Chapter a technique for providing assurance around the identity corresponding to physical devices is introduced, along with discussion of how this can be utilized in the context of the SRUP protocol.

Chapter 7

Command and Control Network Management

This Chapter will examine the issues around how a given physical device, using dynamic device identity as described in the previous Chapter, may provide assurance that it corresponds to a particular logical device identity. This Chapter will introduce a mechanism to solve this problem: and also introduces a number of specific techniques that can be used in support of this. This Chapter addresses Research Question RQ3, and represents the third original element of this research.

Elements of this Chapter have been previously published as [3] and [6], and presented as [5].

7.1 Proof of identity

As described in Section 6.1.4, the use of Dynamic Device Identity necessitates a solution to the problem of mapping a given physical device to whatever logical device identity that it happens to have at any given time. In particular this is required at the time that a given device *joins* a C2 network. If the device's identity can be established at the time it joins, then this can then be maintained throughout its participation within the C2 network.

7.1.1 Simple join

As described in Section 5.6.5.1, the simplest form of join operation is the *simple* or unmoderated join. This makes no attempt to validate the identity of the joining device, and accepts the request based only on the credentials supplied (such as device type) when performing the initial device registration. An example of this is shown in Figure 7.1.

This type of join is only appropriate for situations where there is either no danger of a malicious device being added (such as may be the case in a closed network within a secure perimeter) or for the situation where the overall system cannot be adversely effected by spurious or malicious data being sent from a device. Given the MQTT addressing model, and the use of topic access control (see Section 5.5) there is no significantly greater danger of a malicious device being able to intercept or corrupt the messages to and from other devices (assuming the MQTT broker implementation is bug-free and robust to attempts to employ techniques such as buffer overflow, see Section 2.4.1, to interfere with the operation of the broker).

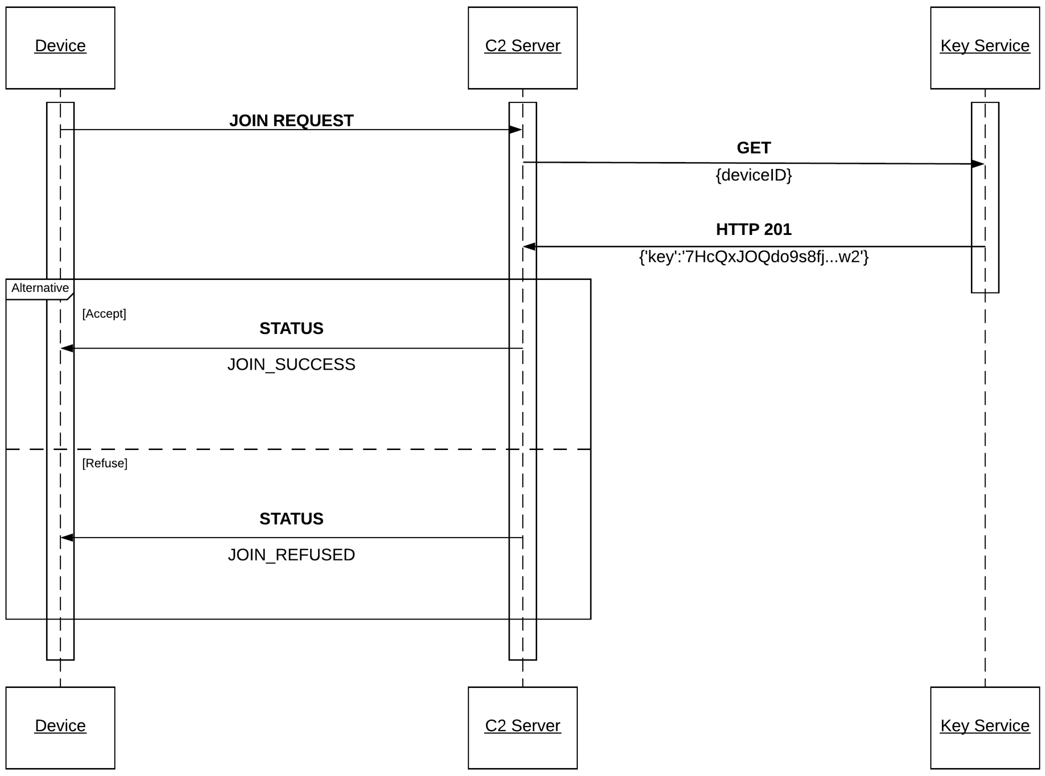


FIGURE 7.1: A sequence diagram showing the message flows for a simple, unmoderated, join operation

7.2 Validating physical identity using third-party observation

The risk from a simple join is that an attacker could stage an attack against the system by intercepting the initial registration request from a device and then registering their own device in place of the real device, since in the context of a simple join there is no validation of the physical identity of the device that has just requested to join.

Addressing this requires a third-party (trusted by the C2 system) in order to perform the validation step by examining the physical device, at the time of the join attempt, to ensure that it is the device which has the identity in question. This observation may be carried out either by a human, or a machine-based, observer. In both cases the process is essentially the same. After receiving a request to perform an observed join, the C2 server will generate a randomly selected, 128-bit, nonce value and encrypt this using the public key associated with the identity in question, and which will then be sent to the device. On receipt of this, the device will decrypt the value (using its private key) and present the data to the observer. This observation will take place *out of band* to the SRUP protocol. The two values will then be compared and the join operation will only be permitted if they match. This technique guarantees that only the logical device involved in the join request is in possession of the correct nonce value, and since the third-party is interacting with the physical device in the physical-world, if the physical device is able to provide this value to the *observer* then the logical device must correspond to that physical device.

Note that since the device would not be part of an existing C2 network at the the time of the join request, the message would need to be sent over an MQTT topic corresponding to the server's ID, specifically reserved for join requests, and to which all registered devices have *write-only* access within MQTT.

A flowchart illustrating the generic observed join process is shown in Figure 7.2.

Although this approach is still theoretically susceptible to an attack scenario where an attacker is able to trigger a malicious device to attempt to join a C2 network, such an approach requires the device to be co-located with the deployed system, and to hastily (and covertly) read the value presented to the observer and relay it to the observer **before** the genuine device. For critical systems, such as those posing a hazard to life (or systems for which an elevated threat is suspected), there would still therefore be a requirement to adopt traditional techniques of manual inspection to determine the authenticity of the physical device, before the join process is initiated; as well as to practice appropriate physical security around the joining process. As always in security, if the attacker has physical access to facilities or other locations, there are a great many other (simpler) forms of attack that they could exploit.

This observation process may be thought of as analogous to the types of physical verification that we are accustomed to in the physical world, for example a photograph in a passport. Although a state-issued document such as a passport is generally regarded to be a means to positively establish the identity of the holder — the mere possession of a passport is insufficient. Rather such documents use a simple to check (but difficult to alter) photograph, and the document is only regarded as valid if the photograph matches (or at least resembles) the bearer. Thus anyone fraudulently obtaining a passport is unable to use it to establish a false identity without modifying it to reflect their appearance (or *vice versa*).

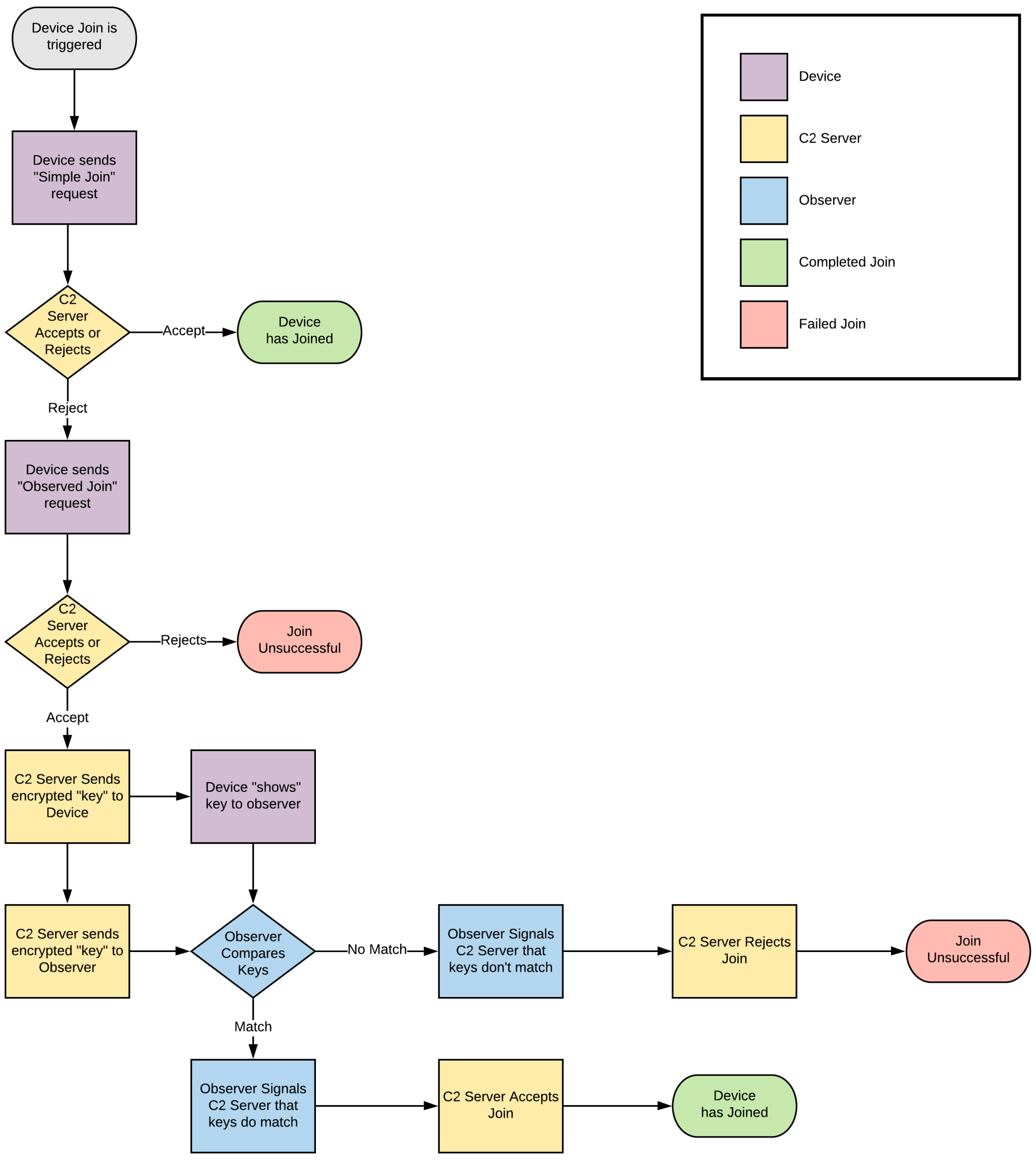

Device Join is triggered
Device sends "Simple Join" request
C2 Server Accepts or Rejects
Accept
Device has Joined
Reject
Device sends "Observed Join" request
C2 Server Accepts or Rejects
Rejects
Join Unsuccessful
Accept
C2 Server Sends encrypted "key" to Device
Device "shows" key to observer
C2 Server sends encrypted "key" to Observer
Observer Compares Keys
No Match
Observer Signals C2 Server that keys don't match
C2 Server Rejects Join
Join Unsuccessful
Match
Observer Signals C2 Server that keys do match
C2 Server Accepts Join
Device has Joined
Device
C2 Server
Observer
Completed Join
Failed Join

FIGURE 7.2: A flowchart illustrating the observed join process, showing the roles of each of the three entities involved.

7.3 Human moderated joins

The human moderated join is the simplest of the two types of moderated join, since it is assumed that the human has easy access to information presented in the C2 system, and is able to view and compare two values without further support from the software. A human moderated join requires that a trusted human operator is present with the device at the time of the join operation, such as might be the case for a sensor (or other device) being manually deployed. Whilst this would likely include most domestic applications, it could also include any commercial, industrial, or military application where the deployment is conducted by-hand.

In this type of join, the join operation is expected to be conducted with relation to a specific pre-known device being deployed or installed, and where the human installer will manually trigger the device to initiate the join process. When the device presents the nonce value for comparison, it is assumed that the human observer is (or is in communication with) the operator of the server, and as such is able to perform the comparison.

A sequence diagram showing the information flow during a human-moderated join operation is shown in Figure 7.3.

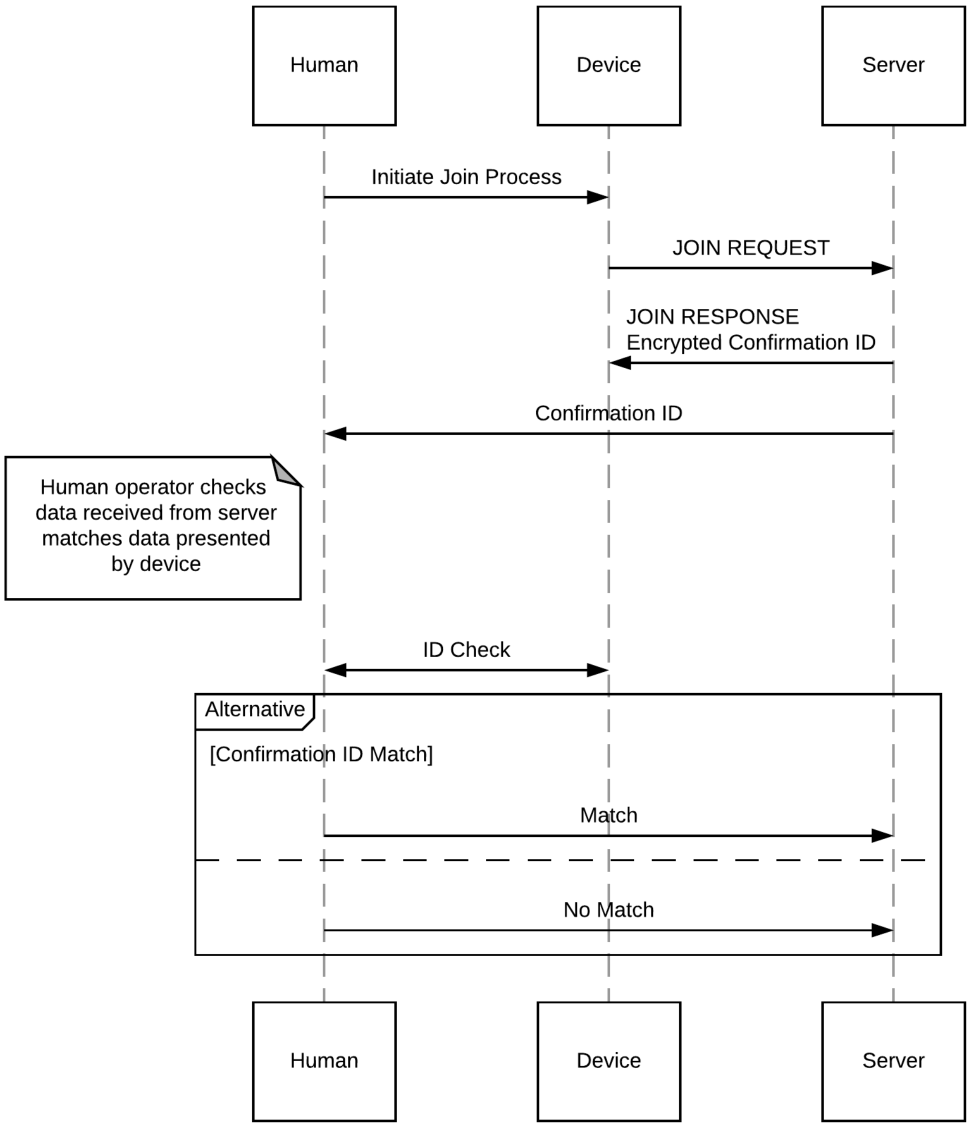


FIGURE 7.3: A sequence diagram showing the message flows for a human moderated join operation

Depending on the device (and the nature of the system, and technical proficiency of the operator) this presentation and comparison can be made in a number of different ways. In the simplest case, the device could display the decrypted numeric value on a suitable display unit.

7.3.1 Hexadecimal notation

Conventionally 128-bit numbers used as UUIDs are shown as a string representation of a 32-digit hexadecimal value, and are typically presented using the 8-4-4-4-12 format described in [234]. Given the visual clarity of this format, it is desirable to adopt the same approach to presenting the 16-bytes of any 128-bit value, including the nonce values used for human observation. An example of a 128-bit UUID used by the protocol, formatted in this manner can be seen in Figure 7.4.

```
C4BF8105-8351-41E0-886F-D25F9A69C5AD
```

FIGURE 7.4: An example of a 36-character string depicting a 128-bit binary value in standard hexadecimal notation.

Such an option is the simplest method to implement. A small 16×2 character, text-mode Liquid Crystal Display (LCD) or Organic Light Emitting Diode (OLED) display would be sufficient to display such a value (without hyphenation), and would likely add only around \$3-4 to the bill-of-materials cost for a device. This approach would be technically compatible with even the most basic 8-bit microcontroller. An example of this is shown in Figure 7.5.

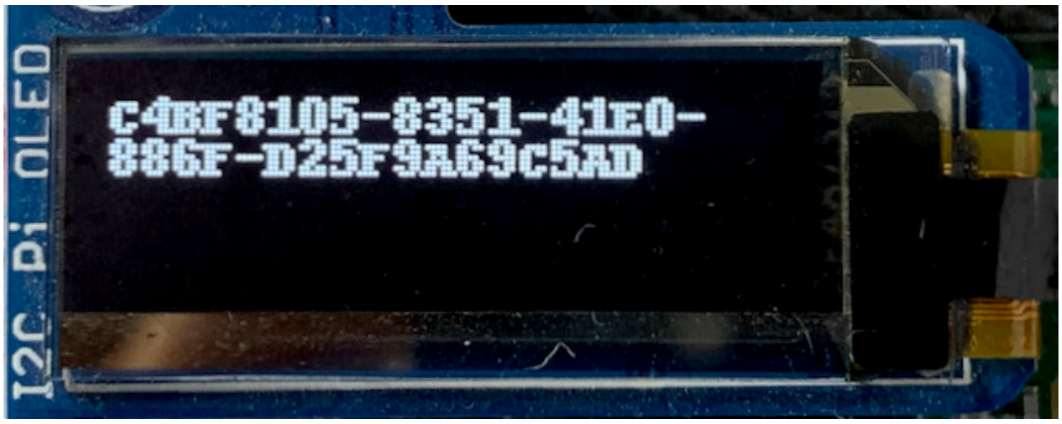


FIGURE 7.5: The 128-bit UUID value, rendered on an OLED screen.

Requiring the careful comparison of two 32-character hexadecimal values is not an especially user-friendly interface, and is really only well suited to environments with specialist users. Research in other contexts [248] has shown that users are not well able to successfully compare values presented in this way.

Instead other options could be adopted to present the value for comparison. Crucially it is not important that the observer be able to identify the underlying value presented to them, but rather only to recognise whether two examples presented match or differ from each other, so alternative methods for comparison may be adopted.

7.3.2 Pictographic representation

An alternative means to present a 128-bit value, is in the form of a pictograph consisting of a 12×12 grid of 144 monochromatic cells. Since such a grid presents 16 additional cells (over and above the 128 required to simply encode the 128-bit value, with each bit denoting the color of a given cell in the grid), the number of cells in *use* within the grid can be reduced, by placing a 2x2 block in each corner.

Such a pictograph would have an appearance something similar to that of a 2D barcode, although one that is designed for human comparison, rather than machine readability. An example of the appearance of such a pictograph is shown in Figure 7.6.

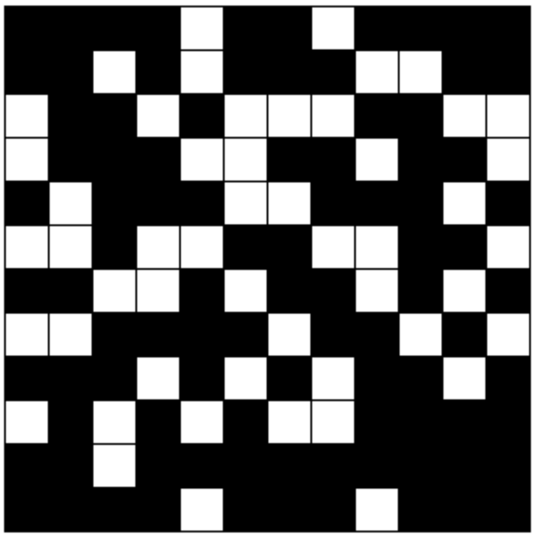


FIGURE 7.6: An example black and white, 12x12 grid depicting a 128-bit binary value.

By analogy with computer graphics, the same number of bits can encode for a smaller 8×8 grid of cells, coloured with one of four colours by using 2-bits per cell. With careful selection of the colours it is possible to produce a grid with sufficient visual difference to be easily distinguished, whilst also avoiding combinations that are indistinct to those with various forms of colourblindness. In particular white, red, blue and yellow were selected since these are still easily distinguishable by patients with deuteranopia or protanopia (both forms of red-green colour blindness), and tritanopia (a rare form of blue-green colour blindness) [249].

An example of using this four-colour approach is shown in Figure 7.7.

For a device with a graphical screen capable of rendering such a pictograph (or which has the necessary hardware to permit its connection to an external display for the purposes of

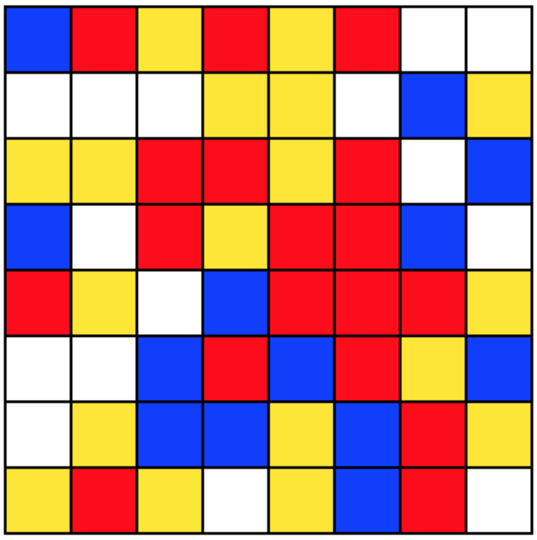


FIGURE 7.7: An example four-colour, 8x8 grid depicting a 128-bit binary value.

validating the join operation) an image of one of these two types could be displayed by the device, and also presented on the user-interface to the server (e.g. a web page) after the registration process had been initiated.

An example device (consisting of a Raspberry Pi 3 fitted with a full-colour LCD graphics display panel) was built, and is shown displaying a four-colour pictogram in Figure 7.8. This device was used in conjunction with an implementation of SRUP using a four-colour pictogram for human moderated joins. The web-based C2 server, which was configured to show a four-colour pictogram is shown in Figure 7.9.

7.3.3 Word-list representation

An alternative to a graphical depiction of the value, would be to present the nonce value in the form of a *phrase* generated by combining words taken from a standard word-list, with the multiple parts of the 128-bit value mapped onto individual words from the list. Similar approaches have been shown for presenting cryptographic keys to users [250], but such a technique can easily also be applied to the presentation of Identity.

A 128-bit value can be divided into ten, 12-bit fragments, with each 12-bit fragment being mapped on to one of a list of $2^{12} = 4096$ different words drawn from a suitably distinctive word-list. A suitable word-list would be one designed for use as a means of providing users a mechanism to generate high-entropy passwords, such as *Diceware* [251]. The remaining 8-bits of the original 128-bit value, could then be represented either by one of 256 additional words; or by using words from within the same list (and using the previously described technique of padding the value with four additional ‘fixed’ bits). Alternatively the technique adopted for the purposes of the demonstration of the capability was to use nine

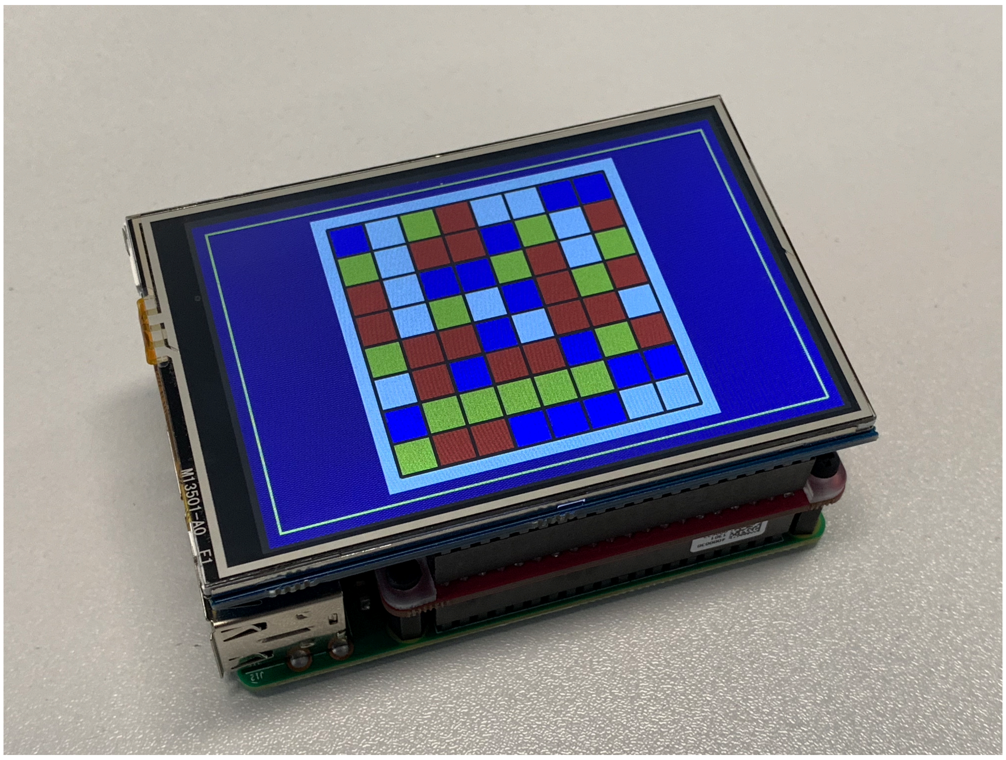


FIGURE 7.8: An example Internet of Things (IoT) device—built from a Raspberry Pi 3 fitted with a full-colour liquid crystal display (LCD) graphics display panel.

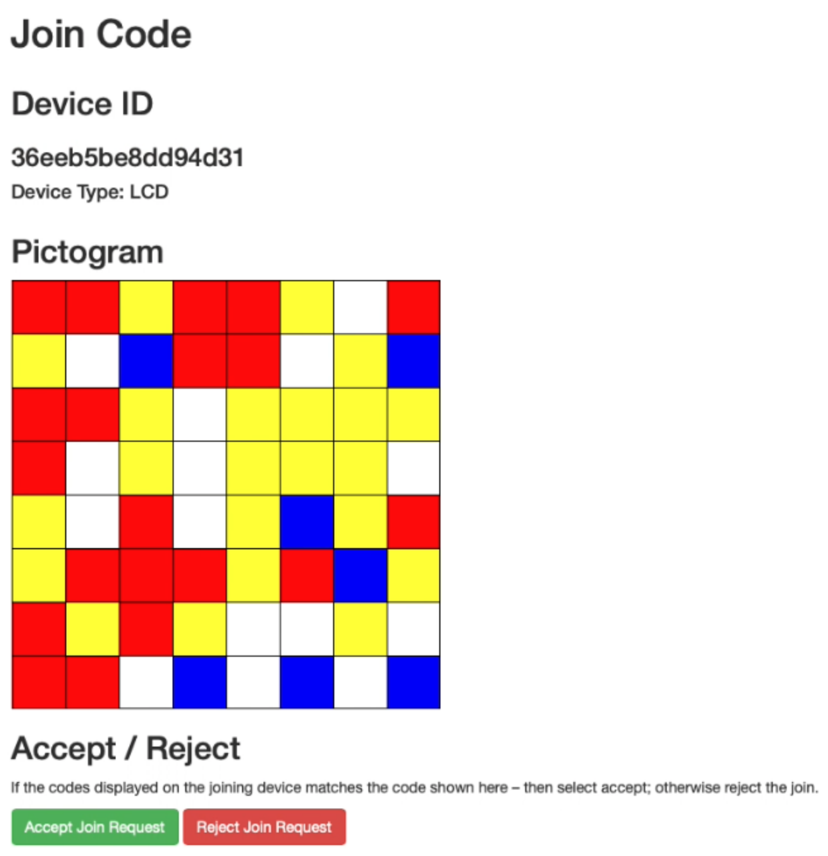
13-bit values, plus an additional tenth 11-bit value (padded to 13 bits). The code used to generate the word list is shown in Listing 7.1.

```python
def wordlist(data):
    bits_list = []
    words = []
    mask = 0x1FFF
    int_value = data
    for i in range(9):
        bits_list.append((int_value >> (13 * i)) & mask)

    bits_list.append(int_value >> 117)

    # Now we have the word list - we need to pad the final block to 13-bits...
    bits_list[9] = bits_list[9] << 2
    for i in range(0, len(bits_list), 2):
        t = [word_list[bits_list[i]], word_list[bits_list[i+1]]]
        words.append(t)
    return words
```

LISTING 7.1: A Python function to generate an identity word list from a 128-bit value

Using this technique the user would be presented with a ten-word “phrase” for comparison to assure identity during the join operation.

An example of using words selected from such a word-list to generate an identity-phrase can be seen in Figure 7.10.



Join Code

Device ID

36eeb5be8dd94d31

Device Type: LCD

Pictogram

Accept / Reject

If the codes displayed on the joining device matches the code shown here – then select accept; otherwise reject the join.

FIGURE 7.9: An example web-based C2 interface for an IoT system, showing an example display depicting a human-comparable display of a nonce value as a four-colour pictogram.

`aggregate`	`diffuser`
`straw`	`zeppelin`
`exponent`	`ungraded`
`problem`	`flail`
`dust`	`postbox`

FIGURE 7.10: An example of an identity phrase generated using a word-list.

The options for presentation of this phrase to the user are similar to those for the presentation of both pictographs and the text strings — although a screen suitable to display simple textual information is simpler than that which might be required to render a coloured pictograph.

As a part of the evaluation of this concept, a device (shown in Figure 7.11) was constructed consisting of a Raspberry Pi Zero W and a three-colour Electronic Ink (eInk) [252] display. This device was used to demonstrate the use of word-list observation.

A video depicting an implementation of the human comparison (showing both word lists, and pictograms) in action can be found at: https://youtu.be/-qBzZ3wT1Tc [253].

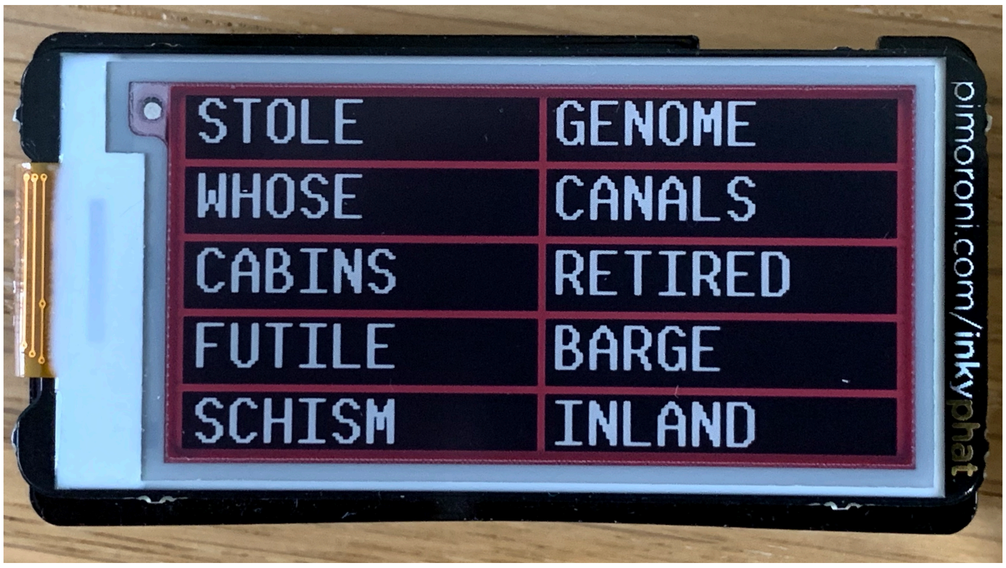


FIGURE 7.11: An example IoT device—built from a Raspberry Pi Zero W fitted with a three-colour eInk display, and depicting a word-list to be used for comparison of 128-bit values.

7.3.4 Other comparison techniques

A number of similar methods have been applied previously [254] to the visualization of complex cryptographic data such as Pretty Good Privacy (PGP) [255] keys, and the 'Visual Host Key' visualization used with SSH (also known as the 'drunken bishop' algorithm) [256]; as well as more complex approaches requiring the combination of two pieces of information, such as visual cryptography [257]. Tan, Bauer, Bonneau, *et al.* even demonstrated the use of unicorn-based graphical presentations of cryptographic keys [248].

7.4 Machine moderated joins

An alternative to using a human to perform the observation and verification steps, is to use a machine-based observer.

This approach is particularly well suited to scenarios with a large number of devices, or where devices are expected to be deployed autonomously without a human presence. For this to work, a trusted *observer node* must be present within the C2 network, and must have already securely joined (for example using a human-moderated observation—as described in the previous Section). This observer node will then utilize one or more techniques to observe the value presented by the joining device, and to confirm that this matches the value that the C2 server had transmitted.

Figure 7.12 shows a sequence diagram depicting the protocol message flows for an machine observed join.

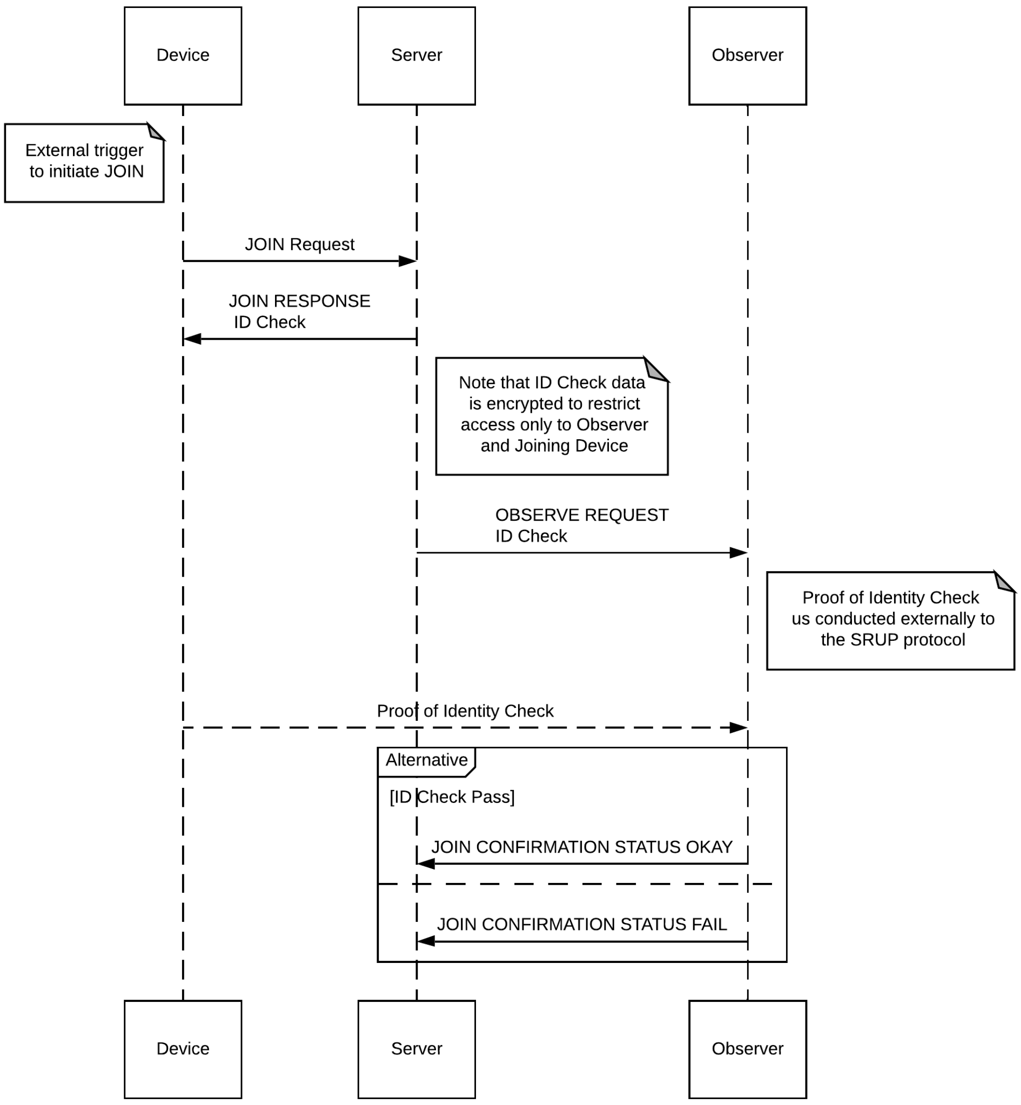

Device
Server
Observer
External trigger to initiate JOIN
JOIN Request
JOIN RESPONSE ID Check
Note that ID Check data is encrypted to restrict access only to Observer and Joining Device
OBSERVE REQUEST ID Check
Proof of Identity Check us conducted externally to the SRUP protocol
Proof of Identity Check
Alternative
[ID Check Pass]
JOIN CONFIRMATION STATUS OKAY
JOIN CONFIRMATION STATUS FAIL
Device
Server
Observer

FIGURE 7.12: A sequence diagram showing the message flows for autonomous, observed, machine moderated join operations

Unlike a scheme adopting a human observer, where the C2 system may be able to directly present a representation of the value to the user for them to compare with the device, an automated observer node must also be securely sent a copy of the code. Specifically, once the C2 server receives an observed join request from a device, the server will respond by sending an `OBSERVED_JOIN_RESPONSE` message to the device, as well as an `OBSERVATION_REQUEST` message to the observer. Both of these messages contain a copy of the 128-bit nonce value to be used for the comparison, and each message is encrypted using the recipient's SRUP public-key, ensuring that only that recipient is able to decrypt the data and read the value.

Once the observer and the device each have their own copy of the nonce, the device should present the value to the observer for comparison and onward signalling.

Although there are a range of technologies that could be adopted to provide this short-range, point-to-point link, this work has specifically examined visual presentation, and short-range RF signalling.

7.4.1 Visual observation technologies

One set of technologies that can be adopted for the observation is machine-readable visual codes—such as barcodes, QR or Data Matrix codes.

Conventional one-dimensional barcodes can store a maximum of around 100 characters (for example Code 128 [258] can store 103 data symbols). However, two-dimensional barcodes such as the QR Code [242], or Data Matrix [259] can store over 1000 symbols.

Since for SRUP observations we need to encode a 128-bit value, any display hardware capable of displaying either a one- or two-dimensional barcode could be adopted — along with any barcode technology capable of displaying 32 hexadecimal characters (e.g. Code-128, Code-93, or Code-39).

The disadvantage of linear one-dimensional codes is that for a 32-character code, the resulting barcode is quite long, and exceeds the convenient aspect-ratio and dimensions of many displays without unduly squeezing the vertical height — resulting in the mark / space size of the code being significantly reduced.

Figure 7.13 shows the same 32-character hexadecimal value, rendered in a number of one-dimensional codes, and Figure 7.14 shows the same data rendered as two-dimensional codes. In all of these cases this is a UUID value, with the punctuating dashes removed to minimize the number of characters required to represent it.

Experimentation using codes for a randomly selected UUID value, rendered on an LCD display, and read using a smartphone camera showed that only Code-93 and Code-128 linear barcodes could be reliably read (and that Code-93, see Figure 7.13b, was most often read). Code-39 format barcodes of the 128-bit value were not readable on the experimental hardware setup

Two-dimensional codes, on the other hand, are already more suitable to being rendered on a display, due to their square shape, and both Data Matrix and QR codes could be easily read by the app, with approximately the same accuracy.

The Data Matrix code has some potential advantages over the QR code, including the resulting size of the code required for a given length of data being smaller, and improved detection and error correction [260].

58D5E212165B4CA0909BC86B9CEE0111

(A) Code-39

58D5E212165B4CA0909BC86B9CEE0111

(B) Code-93

58D5E212165B4CA0909BC86B9CEE0111

(C) Code-128

FIGURE 7.13: A 32-character hexadecimal value, rendered in a number of one-dimensional bar code types.

(A) Data Matrix Code

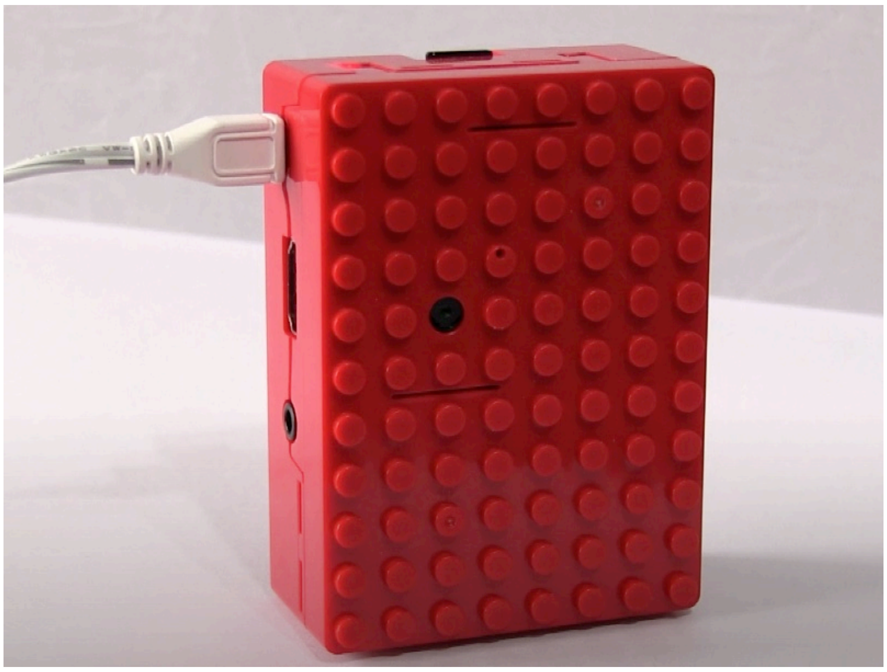
(B) QR Code

FIGURE 7.14: A 32-character hexadecimal value, rendered as two-dimensional bar code types.

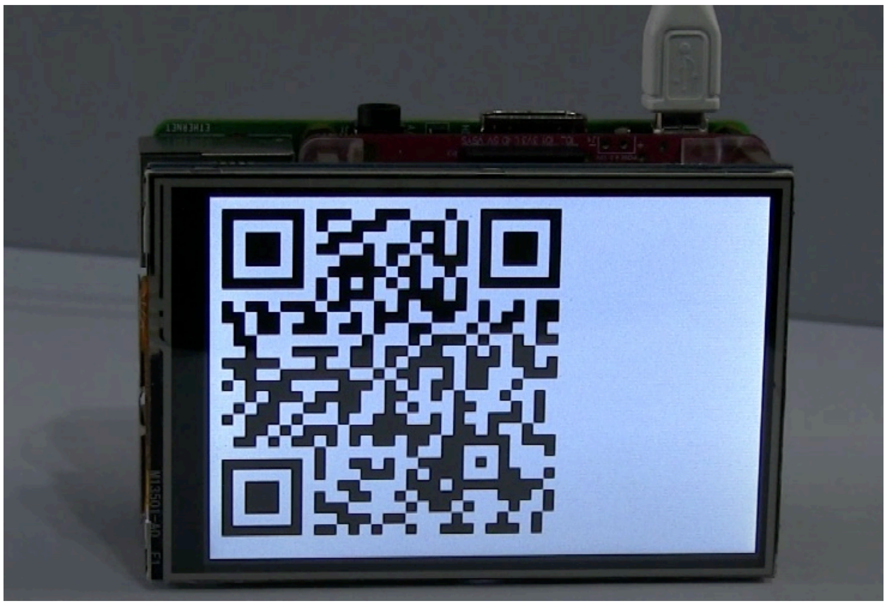
To further explore the utility of these approaches an experiment was created using a Raspberry Pi and camera, looking to automatically locate and extract a 32-character hexadecimal value encoded as a two-dimensional barcode. Unlike the previous example utilizing a smart-phone, here the code was not manually aligned with the reader, rather the software was required to identify where in the image streamed from the camera the barcode was located before it could be decoded.

In both cases the Pi (a Raspberry Pi 3B+), fitted with a Raspberry Pi Camera Module, was running Python code to process the image. For the Data Matrix code, the `pylibdmtx` library [261] was used, and the QR codes were read using the `pyzbar` library [262]. The setup is depicted in Figure 7.15.

In both cases the code was tweaked to maximize performance. With the Data Matrix code, the best performance was achieved when processing a 320×240 grayscale .PNG file (taken from an video capture of the camera, to which the code was being presented) and it



(A) Raspberry Pi with Camera Module



(B) Raspberry Pi with LCD Screen showing a QR Code

FIGURE 7.15: Two Raspberry Pi 3B+ used to assess the performance of a QR code based system for observations. In this example, one is setup as a QR Code reader (fitted with a Raspberry Pi Camera Module); and the other has a colour LCD Screen showing a QR Code.

took an average of 12.7 seconds per image to detect and extract the code. This is in contrast to less than a second when the DM code is manually cropped from the image file. This is unacceptably slow for a system processing video frames, and is in contrast to the `pyzbar` library, searching for QR codes — which was shown to manage to process several frames per second.

Although additional image processing and manipulation techniques could be used (for example to detect and crop the DM code in the image), for the purposes of demonstrating the machine observation techniques, a QR code based approach was selected.

7.4.2 Radio Frequency Identification

In addition to visual observation, a short-range RF link was also adopted to demonstrate a different class of observation node. Although a number of technologies were initially considered (including Bluetooth [263], and the IEEE 802.15.4 Zigbee standard [264]), the best guarantees as to the physical location of the device in question were achieved when using ultra short-range RF communications such as those used for RFID.

RFID technologies fall into two broad classes: low-frequency devices (operating at 125 kHz in Europe), and high-frequency devices operating at 13.56 MHz. [265].

Low frequency systems typically have an operating range of 100 mm, but have very low data transfer capabilities and are only used with simple passive *tags* containing a static serial number determined at the time of manufacture. Such devices are typically used for asset tracking. High frequency systems may store up to 4 Kb of data [266], and may be writable as well as readable. This class of tag is often produced in a *credit-card sized* form-factor, and is often used to provide security access tokens, as well as cashless ticketing in public transportation systems.

NFC also operates on the 13.56 MHz frequency and supports *active* communication between two devices, over a range of a few centimetres.

Despite (or perhaps because of) its ubiquity within the security (RFID) and banking sector (NFC for contactless payments), the state of easily accessible and open-source software to support operations more complex than reading or writing to simple tags is somewhat limited.

Most devices capable of reading NFC data (including most mobile phones) are able to read static tags which have been formatted using the NFC Data Exchange Format (NDEF) standard [267], but there is somewhat limited support for the active data exchange provided by the Simple NDEF Exchange Protocol (SNEP) [268], [269].

To experiment with this technology, a simple device was constructed to utilize SNEP, utilizing open-source example C code and based on the PN532 NFC chipset [270] connected using the Inter-Integrated Circuit Protocol (I^2C). This was interfaced to a Raspberry Pi as a USB serial device, using an Arduino development board, based on the Atmel ATMega32U [271] microcontroller, which is shown in Figure 7.16, and used in conjunction with a simple SRUP implementation.

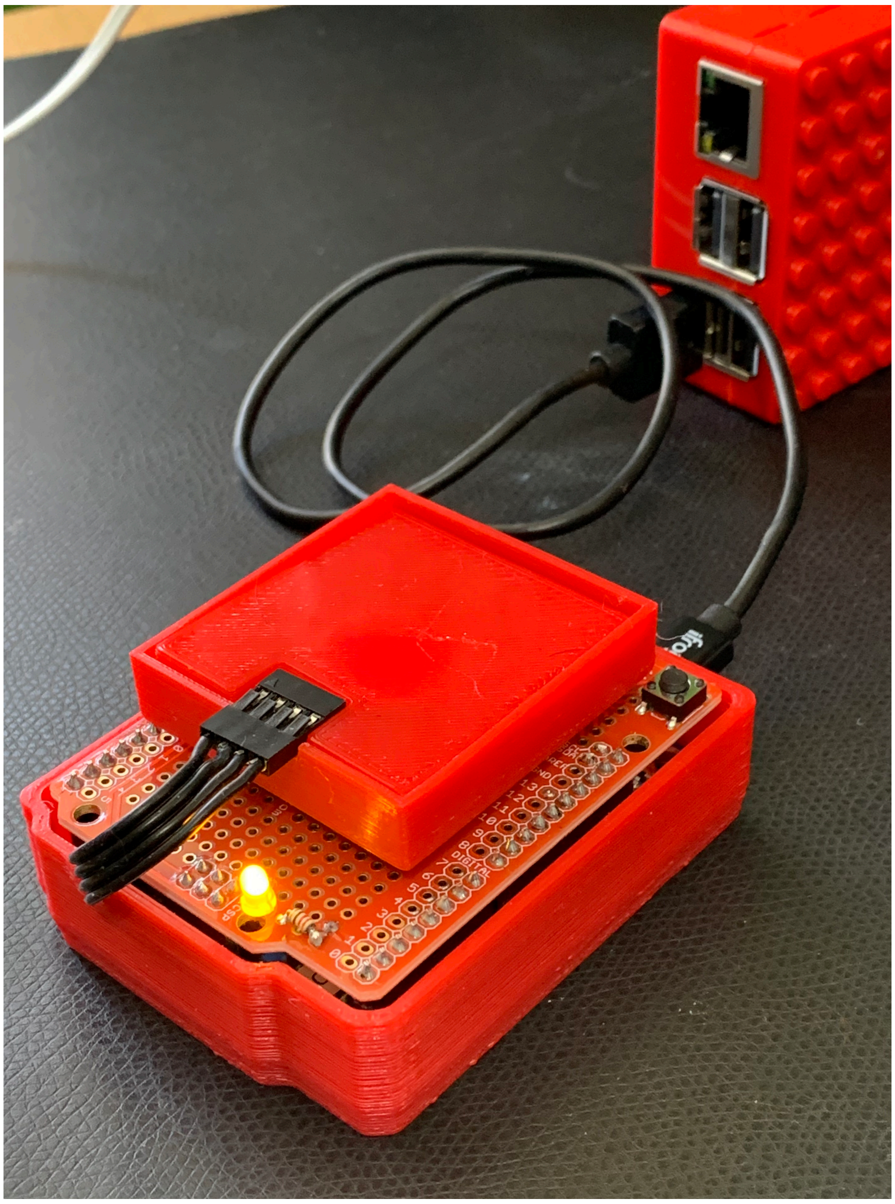


FIGURE 7.16: An near-field communication (NFC) Observer device, consisting of a PN532 NFC module, connected via I^2C to a development board based on an Atmel ATMega32U4 microcontroller, connected to a Raspberry Pi via Universal Serial Bus (USB).

A video demonstrating the machine moderated join process (and depicting both a QR Code based system, and the NFC system) can be found at: https://youtu.be/Vi135raj1LE [272].

7.5 Other machine moderated device identity validation techniques

El-hajj, Fadlallah, Chamoun, *et al.* identify a taxonomy for IoT authentication schemes [273], identifying *Token-based* and *Procedurally based* techniques for device authentication, in addition to identification based around the static *hardware-based* approaches described in Section 6.1.2.

Token-based authentication schemes, such as OAuth2's Device Authorization Grant [274], are typically used to provide a secure mechanism for an individual user to validate their credentials for a given service on a specific device, which is carried out using a third-party device, such as using a mobile-phone, to visit a URL associated with the process. This approach may be seen, for example, when a user logs into a video-sharing website on a smart-television, which presents a QR code for the user to read with their phone, causing the phone's browser to access the page to complete the validation operation.

Although superficially similar, the Device Authorization Grant approach is very different to the one outlined in Section 7.4.1. This approach utilizes the OAuth2 [275] token-based approach and is designed to permit a user to associate a device with the identity that they have defined with an external identity provider (for example, Google, Facebook, etc.). Although a bespoke authorization service could be constructed within the specifications of the OAuth2 standard, this falls somewhat outside of the typical OAuth2 use-case. Additionally, such an approach requires a human-in-the-loop to conventionally log into the authorization service in question, and offers no fully automatic mechanism to pair the physical and logical device identities.

Moreover, using an OAuth-based approach requires that the device (in its deployed state) is able to make outbound HTTPS requests. In contrast, although SRUP does require the device to utilize HTTPS as a part of the initial registration phase (which may be performed prior to operational deployment), the remainder of the join operations take place wholly over the MQTT protocol, which is much more suited to potentially constrained network conditions which may be expected in an operational deployment of a IoT device.

Procedurally based approaches, such as DTLS [199], are utilized for devices adopting static identity and provide security for the messages, but do not themselves offer any guarantee that the physical device in question corresponds to the identity of the logical device and rely on secure on-device storage of the certificate and key.

7.6 Implementing observation-based identity confirmation

A series of devices were constructed, based on the Raspberry Pi platform and utilizing the pySRUP library (see Chapter 9). By using pySRUP, a SRUP IoT device can be quickly written in Python, which can exploit all of the features of the SRUP protocol.

The full information exchange process that occurs during the machine-moderated observed join process (as implemented in SRUP) is illustrated as a sequence diagram in Figure 7.17.

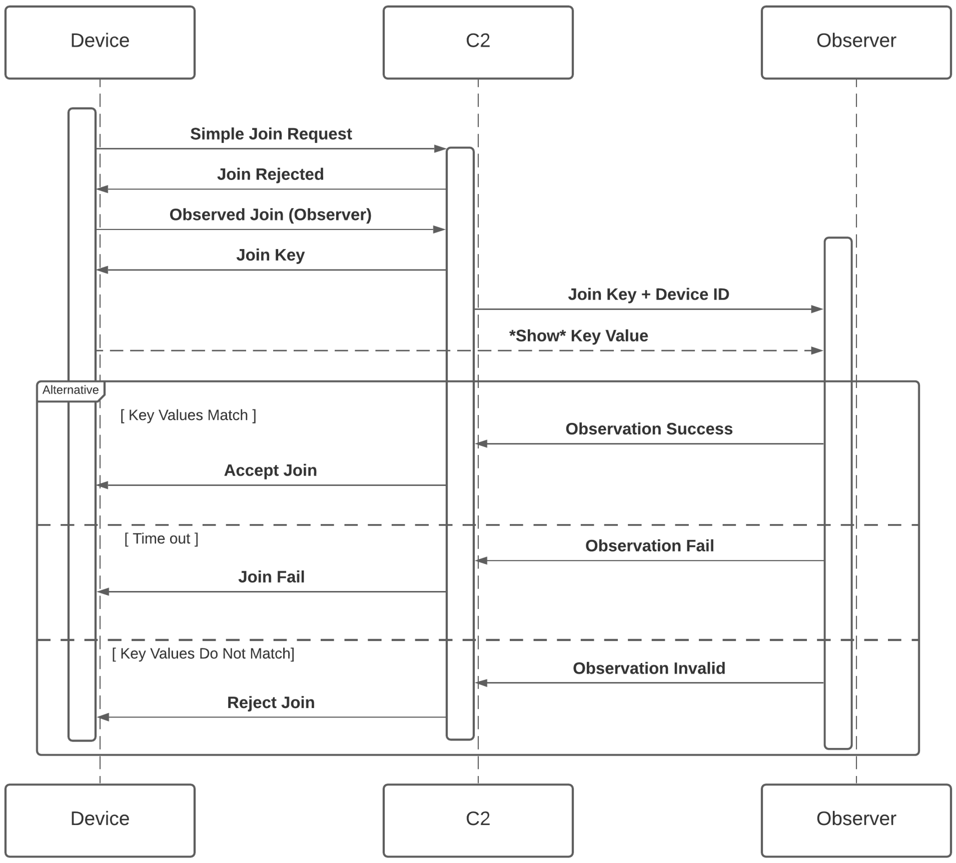


FIGURE 7.17: A sequence diagram showing the machine-moderated join process

7.6.1 Hardware

Two sets of devices were constructed around the Raspberry Pi hardware, one using the visual recognition scheme and one using NFC.

The visual device was identical to the one used for the human-readable pictograms, with a simple observer constructed from a Raspberry Pi fitted with a camera module.

The NFC hardware consisted of two Raspberry Pi computers, each connected to a PN532 module over USB (as described previously). One unit was configured as an NFC observer and a second as a joining device.

In the case of both the visual and the NFC devices, the same pySRUP library code was utilized. The only change required was that the device must specify the required operation in the call-back function relating to the observation and presentation of the identity confirmation.

7.6.2 Operation

With the visual observation scheme, the target device is required to be within the field of view of the observing camera. For the NFC-based observer, the device must be close enough to the observer for the observer to be able to read the data.

In either case, the outcome of an observation attempt is one of three states: either the code was read correctly (VALID), or an invalid code was read (INVALID), or no code could be read and the observation operation timed-out (FAILED). For this example system, in the event that the observer signalled back to the C2 server that it failed to read a value, the C2 server would simply reissue the observation request to both the device and the observer. In a real-world system, the implementation should cap the number of failed read requests by a given device to a (highly system- and implementation-specific) *reasonable number*, but for testing purposes this was not capped in the demonstration.

7.7 Considerations for real-world use of observed join

7.7.1 Human versus machine observation

The question of which type of observer (human or machine), and the larger question of whether an observed join is required at all, is highly dependent on the specifics of the operation and deployment of a given system.

A system utilizing wired sensors on a closed network (such as that found within a larger operational platform, such as a vehicle or factory) may not require any identity validation when joining (especially where physical access to an air-gapped network is controlled). A more typical deployment in a domestic or commercial setting may utilize an open Internet connection to facilitate the communications and as such could use a controlled join. For small-scale deployments, a human-moderated join is an ideal mechanism, especially since this has no requirement for specialist observation nodes. However, a smartphone application could be utilized as an observer (itself joining via a human-moderated join), and then subsequently using either the smartphone's camera for visual recognition, or potentially utilising the phone's NFC capabilities to read values from devices without a user-interface capable of displaying a visual code. Although this implementation has not been explored in this work, the majority of modern smartphones support NFC for mobile payment, and APIs exist within major smartphone operating systems to access some or all of the functionality of the NFC hardware within the phone.

Human moderated joins provide the highest guarantee of device identity, since the human observer is explicitly able to ascertain that the device in question is the device that is being joined (especially where the human installer has manually triggered the join operation themselves). It is however the least scalable to very large deployments, or for scenarios where a large number of join and leave operations are expected to be conducted.

For such large-scale operations, installed devices could join the C2 network, and have their identity validated by a dedicated observation node. For example, within a factory setting an installer could utilize a hand-held observer device using NFC to positively validate devices as they were installed and joined into the system, or for a distributed sensor application, devices could be joined (and validated) as they are deployed.

7.7.2 Benefits of machine observation

In a real-world deployment, different scenarios have different requirements for a machine-moderated join. For devices where the deployment conditions prevent the observer node from physically coming into close proximity to the device (such as an observer covering an area deployment), it may be preferable to conduct an observation via a visual method from a longer distance than may be possible with other technologies. This may require the device to be large enough that both the display, and the device to which it is mounted, are sufficiently visible to the remote observer.

Physical proximity can provide the best guarantee of the identity of the physical device, for scenarios where this is achievable. For example, devices being deployed via a conveyor belt-fed system, devices which are at a known and physically small location (such as passing through a door, or gateway), or devices where a human can physically access and *tap* the device could all utilize this type of approach.

Although designed around the IoT, and the idea of largely static devices, this approach (and the SRUP protocol in general) will work well to support broader classes of *smart* devices, including smart vehicles. However, for the purposes of identity validation, neither of the technologies employed in this example implementation work especially well for scenarios where the device is in motion.

For example, in a scenario where the device would only be in the observers field-of-view for a short period, careful and accurate timing would be required to ensure that the messages supporting the request have been sent (and arrived) with sufficient time for the observer to be ready to view the device. This may require, for example, that the device makes the request some period of time before it expects to be observable (although the longer the time period between the message and the observation, the greater the risk of another device being detected instead).

Similarly, both technologies require some finite time for the read operation to complete (a visual observer would require that the entire display is visible in at least one frame of video, and a Radio Frequency-based observer would require the device to be within the operating range of the reader for at least the duration of the read operation). As such, operations to perform the join may be required to take place during a time period when the device was static (such as at a control point or barrier—prior to entering the smart system's control).

7.7.3 Issues

In the present implementation of the system, a device is required to know the identity of the observer that it wishes to use. In the examples shown above, this is hard-coded into the device's source code. There is currently no *explicit* mechanism within SRUP to specifically enable transmission of an observer's identity to a device. This is deliberate, since until or unless a device joins a C2 network, it is not defined as to which servers may send SRUP messages to it. This is in contrast to the identity of the *default* C2 server for a given device, which is explicitly sent to the device as a part of its initial registration and key exchange process.

Since the device has the SRUP public key belonging to the default server (required so that it can be used as a part of the joining process to validate messages from the server), the server could use the extant `DATA` message within the SRUP protocol in order to send the identity of the observer that should be used, after the server has refused the initial simple join. This could be trivially implemented by a system using the pySRUP library by simply adding a data message handler callback function to the device code and using the existing `send_SRUP_Data` method from the library, to send the observer ID from the server.

7.8 Summary

This Chapter has examined the approach taken to solve the problems associated with the use of dynamic identity within the IoT and described the experimental implementations demonstrated to evaluate these in the context of SRUP. This has shown that both human and machine moderated joining can be used to support the proof-of-identity within an implementation of SRUP. The next and final Chapter in this second part of the thesis will describe how the design of the SRUP protocol addresses and mitigates the security concerns highlighted in Chapter 2.

Chapter 8

Internet of Things Network Security

Chapter 2 outlined a number of cybersecurity risks, and described how they can apply to the IoT. This Chapter will describe how the Secure Remote Update Protocol described in Chapter 5 addresses these risk through the design and implementation of the protocol, and how these address Research Question RQ4.

Section 2.6 introduced three main types of threat to an IoT system:

Threat 1: Compromise of data (or services) from the devices

Threat 2: Attacks against the device

Threat 3: Attacks against connected physical systems

This Chapter will discuss how a number of security risks have been addressed, and in each case the threat will be mapped back against this list.

8.1 Replay attack

For a C2 messaging system, incorporating message signing, one potentially very serious attack vector against a message, is replay attack (see Section 2.6.2.4). The specific risk from replay attack depends on the particular messages being replayed. For example, an attacker capturing a software update message, and after detecting a subsequent update message at a future time may in theory be able to roll the device back to a previous version. Replay attacks primarily fall into either denying service to or from a device (Threat 1), or affecting connected systems (Threat 3).

8.1.1 Common mitigation to replay attack

A number of approaches can be adopted to protect against a replay attack. However, many techniques from other domains are not well suited to applications within IoT C2.

8.1.1.1 Nonce tokens

A common solution to replay attacks is to introduce an additional nonce element within the communication (for example, as used within HTTP ‘Basic’ Authentication [276], [277]) — such that a reply to one message will not be valid when sent in conjunction with another.

By analogy with TCP and the use of tokens within HTTP *basic auth*, in an IoT C2 context this could take the form of a *three-way handshake*, utilizing session and device *tokens*. In this context, messages such as update initialization messages would need to contain an additional element (the session token). A device receiving this would then be required to send a reply containing that session token, plus a device token which it generates. The C2 server would then be required to reply to this message to instigate the action (e.g. a software update), and include both the session and device tokens within the message to validate that the message is not a replay of prior messages. A similar approach is outlined by Feng, Wang, Weng, *et al.* [278].

This approach is not well suited to the IoT however, and goes against the design concept for a lightweight C2 protocol, designed to minimize the communications overhead. It would be very poorly suited for use in situations where devices may have poor-quality network connections, due to the considerable overhead of the additional message traffic.

8.1.1.2 Timestamps

Another commonly adopted solution is to introduce an accurate timestamp within the message [279], which is the approach used by Kerberos Network Authentication [280]. However for this technique to work, it requires that all devices have access to a very accurately synchronized time-signal. Although this could be implemented using Network Time Protocol (NTP) [281] for IoT devices with reliable network connections, it is potentially very challenging if considering communications to deployed devices, over potentially austere data networks.

8.1.1.3 Logging

Another, and much simpler, solution that might be adopted in the context of a C2 protocol, would be to require each device to keep a log of all of the messages that they have previously received. This could either take the form of logging tokens that they have received (restricted to only the tokens contained within validly signed messages to prevent a brute force attack from overwhelming the storage), or by logging a suitable hash of the messages themselves [282]. When using this approach, it would be required that a device checks that any new message received does not match one that has been previously stored. The disadvantage of this approach is that it consumes storage on the device. A 128-bit UUID value such as may be used for the token consumes 16-bytes of storage, and hashing could consume twice this amount of storage per message (assuming SHA-256 hashing at 256-bits or 32-bytes per hash). A receiving device would need to store tokens or hashes in persistent memory in order to prevent the list being reset by forcing the device to power-cycle. Assuming 16-byte tokens, then 512 tokens would consume 8Kb of storage. Whilst this would be trivial for a device with attached storage and a file-system, it would be potentially highly significant for a more simple type of device. This approach also requires that the recipient is able to search the history of previous messages, which further adds to the overhead processing time.

8.1.2 A sequence ID based approach

The method adopted for use within SRUP is a sequence ID based approach, similar in concept to that used within the LoRaWAN protocol for controlling network join requests [283]. This approach has a very small overhead, as it only requires that devices store the most recent sequence ID value that they have received from any given sender. The approach works by requiring that the device compares any newly received message's sequence ID to the stored value for that sender. Only messages that have a sequence ID greater than the stored last received sequence ID from that server may be considered valid. Any message received where the sequence ID is less than or equal to a previously received sequence ID should be discarded.

Adopting this method requires an additional value (a sequence ID) within all messages. A 64-bit (8-byte) sequence ID has been selected, since this would permit the system to send 1,000,000 messages a second for over 58,000 years before overflowing the total possible sequence ID values (see Equations 8.1 & 8.2).

$$\frac{2^{64}}{1 \times 10^{6}} \approx 1.845 \times 10^{12}\text{s} \tag{8.1}$$

$$\frac{1.845 \times 10^{12}\text{s}}{60 \times 60 \times 24} \approx 2.1^{7}\text{days} \approx 58,000\text{years} \tag{8.2}$$

In order to conserve space in the message, it might appear to be preferable to adopt a smaller size (e.g. a 6-byte sequence ID — which would still permit 100,000 messages / second for 89 years). However since a 6-byte integer is not a standard size it would be required to implement a bespoke 6-byte integer type to handle this value, and although this could be easily done in most languages, given that for even the smallest message type the difference would make $< 1\%$ difference to the overall message size there is no value in doing this versus using a standard 64-bit integer (e.g. `uint64_t` or `unsigned long long` in C) for the Sequence ID.

The implementation would work in the same way for messages sent by the server, and messages sent by the devices. In all cases, the sender would simply store the last value it used to communicate with that receiver, in local persistent storage, and increment it by one for each new message.

Since under the hierarchical C2 paradigm used by SRUP, devices can only accept messages from a C2 server devices would be required to store (within persistent storage such as a file system or EEPROM) just one such value for each server that the device receives messages from, and one for each server it sends messages to. The relatively small-size of the sequence ID means that even devices implemented using tiny microcontroller-based hardware, would be unlikely to face storage limitations. For example just 1kB of storage is more than sufficient for 128 sequence IDs at 64-bits (and it is asserted that a requirement for a device to be in active communications with 64 different C2 servers at any one time is highly unlikely).

On the server side, it can be assumed that storage will be much less of an issue. This approach would require that a server keeps a last Sequence ID from all devices that they have communicated with (until such a time that the device has been removed from the system). Even a server with 100,000,000 devices would require less than 800MB of storage for their current Sequence IDs, a relatively trivial amount for any cloud-based server implementation.

8.2 Message spoofing

The use of cryptographic signatures to protect messages means that it is extremely difficult for an attacker to create a spoof message that would not be rejected by the receiver. Spoofed messages are primarily a threat to connected systems (Threat 3).

Once a message has been assembled from its constituent fields, the SHA-256 cryptographic hash function is run to generate a secure message hash. This hash is then signed using the sender's private key. Subsequent validation of the message requires the receiver to check that the signature received corresponds to the hash value received when decrypted using the sender's public key. If any part of the data is altered, it will cause the signature to fail to validate against the content of the message.

There are two possible attacks versus this scheme. Either the attacker needs to break the algorithm used to sign the hash — to create a new message with a different hash, but still appearing to originate from the original sender; or to break the hashing algorithm to generate a new message with the same message hash. In both cases there are no known mechanisms to achieve this. The RSA cryptosystem is generally regarded as secure [284] as are modern implementations of the Secure Hash Algorithm (SHA) — such as SHA-256. There have however been a number of attacks versus earlier versions of SHA, and SHA-1 was shown in 2017 to be broken, such that two non-identical documents could yield the same hash [285]. As such, use of a more secure algorithm such as SHA-256 is required to prevent this type of collision attack.

8.3 Attacks against MQTT and C2 systems

For systems where there is a requirement for message confidentiality, the use of MQTT over TLS means that data in transit is well protected against eavesdropping and interception (Threat 1). Therefore in order to attack the system so as to be able to obtain data from devices or sensors, it would be necessary to attack either the broker or the C2 server itself.

8.3.1 MQTT broker attack

One potential weak-point of any system built around the MQTT protocol is the broker, however the OSS implementations of MQTT broker software are well-regarded in terms of security and are the subject of active security research [286]. In particular though there is a threat to the broker from DoS attacks both from malicious publishing of large volumes of large sized MQTT messages to overwhelm the broker, and from more generic packet flooding attacks [287], such as TCP synchronize or SYN flooding.

Of these approaches Firdous, Baig, Valli, *et al.* show the more disruptive is the publish flood. However, the use of a certificate-based TLS encrypted MQTT connection (see Section 5.5) would somewhat mitigate against this by restricting broker access only to registered devices, and thus any nodes being used to stage such as DDoS being required to have established an identity with the Key Exchange service (see Section 6.2.1) prior to joining the attack. Although SYN flooding [288] is widely seen versus HTTP its impact on MQTT was shown to be minimal, apart from the risk to disrupting legitimate traffic to the broker by choking the broker’s incoming bandwidth [287]. Research has shown [289] that machine-learning approaches to DoS attack detection and protection can also be adopted to minimize this threat.

8.3.2 C2 server attack

The other potential area of attack against the data within a C2 system, is an attack against the C2 server itself.

The use of the server token file (see Section 6.4) ensures that only approved servers can operate within a given SRUP universe. This ensures that devices cannot be *hijacked* by a malicious C2 server, preventing both man-in-the-middle attacks and a scenario equivalent to *deauthing* [290] devices in a given C2 network.

Although this work describes a reference implementation of a C2 server (see Chapter 9), the detailed specifics of any real-world C2 server implementation are inherently application specific. It is anticipated that the C2 server would be implemented using a cloud-hosting service such that it was Internet-facing, and that the implementation would provide a web-application front-end. This approach requires only that the user of the C2 service had access to a compatible web browser, obviating the need for a bespoke client application. However a hybrid approach utilizing a REST API which could be consumed by a desktop, mobile, or third-party web application, could also be adopted where required.

User authentication in this context could adopt standard techniques for web authentication — of a type very generally used across the Internet for web applications.

Given the potential to attack the back-end services, it is also necessary to secure the cloud hosting platform against attack. This too is, however, a well understood issue and is in common with the in-service deployment of any real-world web application. Providing user authentication, and the security of the servers on which the application runs, are not breached the C2 server may be regarded as secure to attacks.

8.3.3 Attack of observer nodes

A third type of attack which falls under Threat 1 is the addition of a malicious node within a C2 system. Although techniques to *observe* join operations (see Section 7.2) protect against this type of attack, there is the potential to stage an attack against an automated observer node. However the use of additional payload encryption for the observation request message (see Figure 7.2) makes a direct attack on the process somewhat infeasible. There is however a potential for a malicious observer to join the system, if not correctly protected. If a malicious observer was present within the system, then it could enable the addition of a malicious device to a C2 system. To prevent against this, it is essential than any observer nodes present within a system are themselves joined via an observed join (e.g. a *human* join — although in principle one observer node could add another, the first such node in a system would always require a trusted human observer for its deployment).

8.3.4 Crypto-agility

As described in Section 5.1, the protocol is designed independently of any specific cryptographic algorithm or implementation. As such, although a given implementation of SRUP will necessarily adopt a particular cryptosystem (as a part of both protecting against Threat 1 and Threat 3), the underlying protocol's independence from this means that the protocol (and, to a slightly lesser extent, any given implementation of it) demonstrates crypto-agility. As such if at some future time the RSA protocol is broken, either as a result of quantum computing (see Section 2.8.4) or any other discovered attack, a new implementation of SRUP could easily be built to utilize any other asymmetric protocol which supports both message encryption and message signing.

8.3.5 Physical attack

In the event that an attacker is able to gain physical access to the device it can be assumed that, unless the device has been built to utilize encrypted storage and a trusted boot process [291], the attacker would be able to gain access to any data stored on the device (Threat 1), as well as installing malware on the device (Threat 2), and be able to interfere (physically or electronically) with any connected equipment (Threat 3). As such it is essential (in any real-world deployment of an IoT system) to either provide physical security for the devices to prevent them falling into the hands of malicious parties, or to protect them electronically through the addition of trusted boot (and ideally, both).

For a system built using SRUP, having physical access to a device also equates to gaining access to the device's private key (as well as the server's public key). Since the server's public key is by definition publicly available (on demand via the registration process) we may largely disregard this as being of use to an attacker. However having access to the device's private key means that (by the definition used in Section 6.1) the attacker may take over the identity of the device. As such any messages that the system would permit the device to send, could be spoofed and be sent by any other device that the attacker may devise (and assign the original device's identity to).

Although a system based around the principles described here, makes no attempt to provide a mechanism to monitor or detect such device *identity theft*, the centralized nature of the C2 server paradigm, with no peer-to-peer messaging, means that the potential damage that can be caused by that identity theft may be contained to the server. Other devices on the C2 network can not be directly affected. Given that (by definition) devices are always subordinate to the server, there is limited scope for damaging the overall system via spoofing device messages, beyond that which may be caused by directly interfering with the device's sensors (which would also be possible in the event that the attacker had physical access to the device).

In the event that such an attack were detected, the device can be centrally removed or deregistered from the network, to revoke its ability to communicate further on the system.

Although such a device may be able to re-register and gain a new identity (subject to any registration access credentials not also having been revoked), in a human-moderated system it would not be able to join the server without the assistance of a compromised (but trusted by the system) human. This *insider threat* situation is one which almost no security paradigm would be able to protect against. The greatest threat to any centralized C2 system is compromise of the centre, however the asymmetric cryptographic approach described means that compromise of a subordinate node has no impact on the security of the central server, and the likely cloud-based nature of the C2 server means that generic (and industry-standard) best-practice defences can be applied.

8.4 Software update

One of the primary mechanisms to reduce the likelihood of malware infections for IoT devices, and therefore addressing Threat 2, is a secure software update process.

The security involved with such a process is three-fold. Firstly it is important that only an authorised party can trigger a software update operation in the first place. Since only a C2 server can instigate an update, spoofing an update message (either to attempt to get the device to load a malicious software update; or to roll-back a previous update; or to cycle a continuous update to cause a DoS against the device) is protected against using the mechanisms discussed elsewhere in this Chapter.

The second part of the update process is to ensure that when the device acts in response to an authentic update message, that it retrieves the correct and unmodified data. This is protected against within SRUP via the fields within the SRUP update messages (see Section 5.6.1). The addition of a SHA-256 hash for the software to be downloaded, all-but guarantees that the device is able to validate that the software it obtains is the software payload that it was intended to receive. The use of a HTTPS connection to retrieve this software (combined with authenticated access, with the credentials passed to the device previously via other C2 messages) also ensures that the origin of the software cannot be spoofed, and that the data cannot easily be obtained by a third-party for purposes such as reverse engineering.

The final part is that the process of obtaining a software update is separated from the process to activate that software on the device. The SRUP protocol requires that the devices confirm reception of the software to the C2 server (either confirming successful retrieval of the software, or signalling a failure caused by either the message hash not matching; the file not being available on the server; or that the server could not be accessed). This positive confirmation process ensures that a C2 server can track whether and when the devices are ready to activate the new software. The C2 server can then send activate messages to the device(s) to trigger the new software to be installed / deployed / etc. (although the details of how the device actually executes this update, are application and device implementation specific). This additional step should ensure that all devices can switch at the same time and so avoid a situation where multiple versions of the device software are live on the network at any given time.

Finally the addition of the ID Request message (Section 5.6.3.1) means that the C2 server can positively confirm the version of the software being run on any given device. (Although the details of the content of the Device ID message are implementation specific — it could easily include details such as the SHA-256 hash of the software that the device is running).

8.5 SRUP and the DCMS Code of Practice for IoT security

Section 2.6 described the thirteen design principles of the NCSC / DCMS *Code of Practice for Consumer IoT Security* [125].

Not all of these are applicable to a protocol such as SRUP; but items 1, 3–7 are relevant to this research.

1. No default passwords
3. Keep software updated
4. Securely store credentials and security-sensitive data
5. Communicate securely
6. Minimize exposed attack surfaces
7. Ensure software integrity

SRUP does not use passwords for device security (although a real-world implementation of a web-based C2 system would likely use a password-based user authentication scheme), and the dynamic identity and key distribution system described in Chapter 6 address this requirement (1) well.

The software update messages within SRUP are designed to both facilitate easier remote software operations, and provide assurances about software integrity via integrated message hashing and validation (3, 7).

All SRUP messages can be encapsulated within TLS encrypted MQTT, satisfying the requirement for secure (encrypted) communications (5).

The use of MQTT was selected in part because it eliminates the requirement for a device to have other ports open for message traffic (6).

Only recommendation 4 is not directly implemented as a part of this work. However, if running on suitable hardware, SRUP could easily be implemented to make use of TPM or other trusted-execution environments for secure boot, and storage of cryptographic keys.

8.6 Summary

This final Chapter in the second Part of the thesis has described how the SRUP protocol has been designed to mitigate the cybersecurity risks described previously in this thesis. The third Part of the thesis will describe the implementation of the protocol and supporting technologies as described in this second Part, and will explore how together these address the final three Research Questions. The next Chapter (Chapter 9) will describe the reference implementation of SRUP in the form of a software library, and discus how this has been optimized for ease of use by developers wishing to utilize it to build their own SRUP-based IoT systems.

Part III

Implementation & Experimentation

Chapter 9

Implementing the Secure Remote Update Protocol

The third and final Part of this thesis describes the implementation of SRUP and experimental assessment of its performance. This Chapter describes the software library implementation of SRUP, its associated backend key-exchange service, and other related tools. This Chapter describes the work to answer Research Question RQ5.

All of the original software described in the Chapter is *Open Source Software*, and has been released [292] under the terms of the MIT Licence[1].

Elements of this Chapter have been previously published as [4], and presented as [5].

9.1 SRUP library architecture

In order to combine the best elements of a binary implementation, and the ease of use of a scripting-language, a hybrid approach was adopted for the implementation of SRUP. For the purposes of this research, the binary code was written in C++, and Python was the scripting-language selected.

The design concept was to produce an underlying implementation to generate and process the byte-stream to be used as the MQTT message payload. This code was implemented using C++, and formed a binary library (`libSRUP_Lib`). Using this approach ensured that extant OSS binary library implementations of cryptographic functions (such as RSA signatures) could be efficiently included by incorporating these libraries (e.g. `libcrypto` — a part of OpenSSL[2]) into the build process.

[1] https://mit-license.org
[2] https://www.openssl.org

A Python wrapper (`pySRUPLib`) was implemented to permit direct utilization of the binary library implementation of SRUP within Python, and this was then itself wrapped with a *pure* Python library (`pySRUP`) consisting of classes intended to be called directly from a user's application code. This Python wrapper was designed to implement as much of the generic SRUP functionality (such as ensuring valid sequence IDs) as possible, and thus greatly reducing the implementation (and therefore detailed understanding of the protocol required) to build an application using it.

An architecture diagram illustrating this approach can be seen in Figure 9.1.

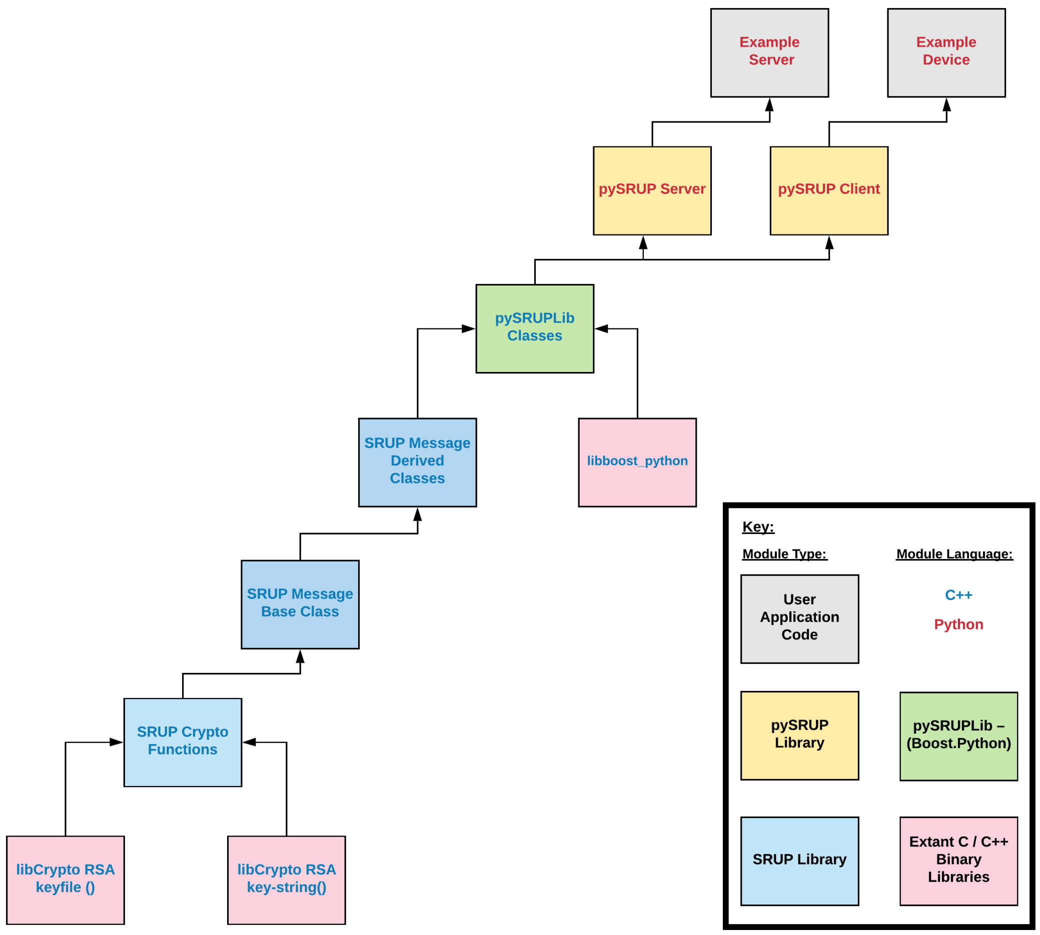


FIGURE 9.1: An architecture diagram illustrating the combined C++ / Python implementation. The colour of the box denotes the type of the element, and the colour of the text denotes the programming language used for this element.

Specifically, the following elements were implemented as a part of this work:

1. C++ library (`libSRUP_Lib`)
2. Binary Python library (`pySRUPLib`)
3. Pure Python wrapper class (`pySRUP`)

4. Key exchange server
5. Example web-based C2 system
6. Containerized backend
7. Bootstrapping key generation tool

Each of these will be described in more detail, in the following Sections.

9.2 C++ library

The core of the implementation of SRUP has been the C++ implementation of `libSRUP_Lib`. The implementation adopted an object-oriented approach — defining an abstract base class for SRUP messages, and subsequently deriving all of the concrete message classes from this.

A Unified Modeling Language (UML) class diagram for the classes defined within the library is shown in Figure 9.2.

The implementation makes use of C++ inheritance in order to reduce the complexity required within each message class. All messages inherit from a base message class, and therefore can utilize the implementation of the methods and properties within that base-class. Two other abstract classes are also defined: one to provide an implementation for all of the message types that do not add any additional fields to the base message (`SRUP_MSG_SIMPLE`); and one for the observation messages which need to include additional encryption elements (`SRUP_MSG_OBS_BASE`).

To avoid making assumptions about the data sizes of particular C++ types on any given platform that the library may be built for (given that the C++ standard makes only minimum length guarantees [293]) explicitly sized data types were used throughout (e.g. `uint16_t` used to store 2-byte length elements).

C++ was selected as the development language for this library due to the ease of manipulating low-level data structures which C++ shares with C, and because of the modern language's performance characteristics. It is also a language for which there exist easy bindings to build Python classes, using (for example) Boost.Python [294]. However, although C++ is a very widely used language, it is a language where there are frequently cited significant concerns over memory safety (for example [295]).

Therefore were SRUP to ever be adopted in any safety-critical applications, the protocol could be reimplemented in a memory safe language such as Rust, which has been shown to be verifiably memory safe [296] [297], when used correctly [298].

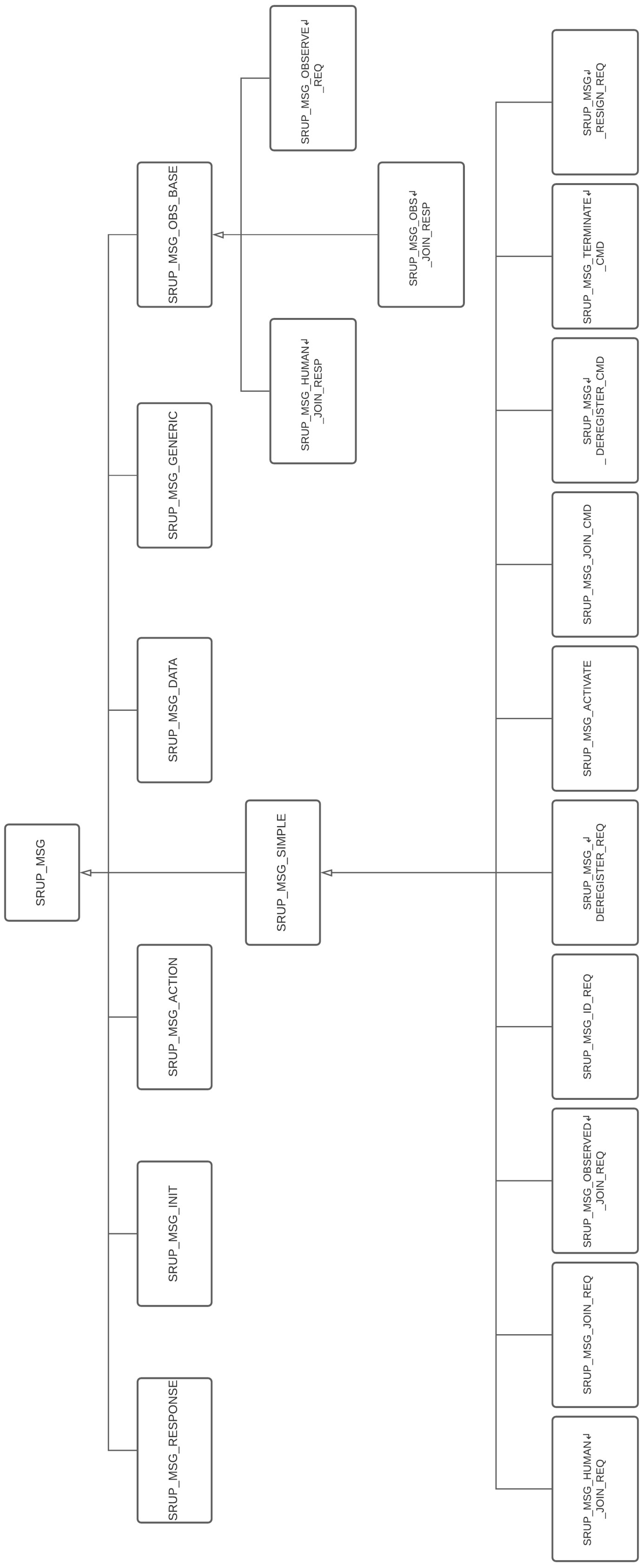


FIGURE 9.2: A UML Class diagram depicting the hierarchical class structure for the SRUP message classes, as defined within the C++ implementation of the `libSRUP_Lib` library.

When reassembling the messages from the byte-stream the C++ library will first decode the message using the generic message class. This will only attempt to use the first few bytes of the message — but will enable the calling code to identify which type of message the byte-stream contains. Once this has been determined, the message can then be demarshalled into the correct message class.

The SRUP data messages do not contain any specific indicators as to the type of the data they contain — beyond the (application specific) data ID value. Therefore although the data message will contain a byte-stream relating to the data, the data message class is not able to determine the type of that data. Consequentially the class implements a number of *getter* functions for the data property of a message, with each (attempting to) return the data using the specified type. The calling code must know the type of the data associated with the data ID therefore, and use the correct method when accessing the data.

9.3 Binary Python library

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 dependencies 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. The performance of modern hardware is sufficient that the *cost* in terms of the speed of execution of running *slower* software is no longer a consideration. By 2000, (the then still nascent) Python was approaching comparable performance to compiled code for some real-world tasks [299], and by 2020 Python was widely regarded as the language of choice for *data science* and other data processing tasks [300].

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 application software by directly using machine code or assembly language, and increasingly intermediate languages (such as C or C++) are restricted to being used only for *systems* programming, or applications for which the speed of execution is critical.

9.3.1 Why Python?

Given the prevalence of Python in the current software trends (Python is regularly ranked as the ‘top’ programming language in surveys of real-world language use [301]), 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 the lower-level work (such as marshalling & demarshalling bytes, and calculating & checking cryptographic signatures) can be performed in a language optimized for speed and efficient byte-level operations on data. Extant MQTT and HTTP libraries (such as Paho[3] and Requests[4]) make it extremely easy to interface with the backend systems from Python.

9.3.2 Calling C++ from Python

A number of approaches exist to call C & C++ code from Python. These range from using Python's built-in `ctypes` library and a suitably built C or C++ library [302], using Cython to build the C / C++ within the Python code [303], or using a dynamic approach to call a separately executing binary via interprocess communication — such as Apache Thrift [304].

The selected approach was to utilize the Boost.Python C++ library [305], to build a native binary which can be called directly from Python, and which fully supports language features from both C++ and Python.

The resulting `pySRUPLib` library provides a Python class for each of the (concrete) C++ SRUP message classes, utilizing a similar approach to the class hierarchy to maximize reuse by exploiting inheritance. This library added no additional functionality over the C++ implementation of `libSRUP_Lib`.

9.4 Python wrapper class

The third element of the implementation was specifically designed to address RQ5, and the question of making SRUP *easier to use* than implementing an alternative approach.

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.

Ideally any solution would enable a device developer to utilize secure communications without needing to be exposed to the specific details of the security process and its implementation. A measure of success in this regards therefore, is whether a device can be implemented in less code than would be required for an unsecured solution.

[3]https://pypi.org/project/paho-mqtt/
[4]https://github.com/requests/requests

In order to achieve this, the pySRUP library was created. This library is wholly written in Python (making it what is also known as a *pure* Python library), and interfaces with the `pySRUPLib` binary library. However, in addition to the simple implementation of the SRUP message classes, pySRUP wraps in all of the functionality to interface with the MQTT broker (via the Paho library), and all of the registration and key exchange services (utilizing the Requests HTTP library). The library implements three classes: one for a device, one for a C2 server, and one supporting *syndication devices* (see Chapter 10).

Providing the necessary backend is in place (see Section 9.6), the only thing that a device developer needs to know is the URL for the key exchange service associated with that backend. All other aspects of the device registration and configuration process are taken care of automatically by the pySRUP library code, and subsequently saved to a (user-specified) configuration file for future use.

In order to permit the device specific code to be triggered on the basis of different SRUP messages, pySRUP supports the use of callback functions to be used on receipt of various message types (once they have been validated by the library). For example, a developer wishing to construct a device which can handle a SRUP *action* message can simply write a function defining the behaviour of the device required for a given action type. This is passed to the object instantiated from the pySRUP device class, in the form of a callback function. 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 the sequence ID). If the message passes the validation tests, then the details of the message will be passed to the handler function to perform the correct action.

For the more complex process around the software update operation, the library can automatically handle the intermediate message exchanges, notifying the user application only when the retrieval and validation of the update data has been completed, and the *activation* signal has been received.

9.4.1 Ease of use comparison

In order to demonstrate the simplicity of using pySRUP, a simple example IoT device was constructed from a Raspberry Pi Zero W and a custom circuit-board with a number of Light Emitting Diodes (LEDs) to represent different actions that the device could perform. This example device is shown in Figure 9.3.

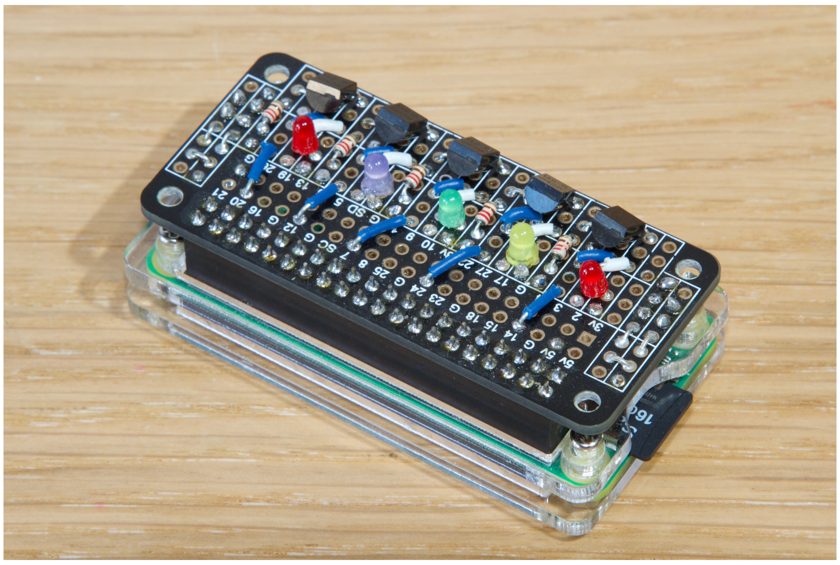


FIGURE 9.3: A photograph of an example IoT device, consisting of a Raspberry Pi Zero W and a custom circuit board, used as a part of an experiment to examine the the ease of use of the pySRUP library.

The device software, written using the pySRUP library, included handling SRUP action and data messages, as well as a simple implementation of remote software update. This software consisted of less than 100-lines of Python code. Listings 9.1 & 9.2 illustrate the brevity of the code required to implement the behavior of this device.

Inspecting the code more closely, it can be seen that Listing 9.1 contains the Python *import* statements, and the definitions of the functions determining the actual behaviour of the device, and only Listing 9.2 contains the pySRUP specific code to trigger that behaviour based on the SRUP messages received. Note that this example also includes a simplified remote software update. All of the code to process the update initiate message, retrieve and validate the software files, and signal back to the C2 server has been automatically handled by the library in this case.

When utilizing the pySRUP library, a developer is required to write significantly 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 for 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 function would be required to identify the correct behaviour to perform. Both of these elements require considerably more code and more consideration than an approach based on pySRUP.

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

FILENAME = "device.py"
DELAY = 0.75
LED_STATE = False
auth = ("AJP", "Password!")

def led_setup():
    # Set LED pins to output & their state as off...
    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):
    # Toggle state of one of the LEDs
    if not state:
        GPIO.output(24, GPIO.HIGH)
        return True
    else:
        GPIO.output(24, GPIO.LOW)
        return False

def switch():
    # Switch LED on for DELAY msec...
    GPIO.output(20, GPIO.HIGH)
    time.sleep(DELAY)
    GPIO.output(20, GPIO.LOW)
```

LISTING 9.1: The device-side Python code for a simple IoT application, using the SRUP protocol to control a device: part one — preamble & GPIO setup.

```python
def on_action(msg_action):
    # Callback function for SRUP ACTION messages
    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.dat")
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()
```

LISTING 9.2: The device-side Python code for a simple IoT application, using the SRUP protocol to control a device: part two — SRUP Specific Code.

9.5 Web-based C2 system

An example implementation of a web-based C2 system was also produced as a part of this work. This too was implemented in Python, since the pySRUP library (as described in Section 9.4) could be utilized. The C2 system instantiates the *Server* class from pySRUP. For this example the Flask[5] library was utilized to provide the web development framework, although any other framework could very easily be substituted. Flask [306] was adopted, since it provides a very lightweight web development framework, supporting Jinja[6]

[5] https://flask.palletsprojects.com/
[6] https://www.palletsprojects.com/p/jinja/

templates for dynamic page generation. Any C2 system for real-world use is inherently application specific, so this example implementation is included only as a reference for developers implementing their own bespoke systems.

Setting up a C2 server requires some additional steps when compared with setting up a device, not-least when it comes to setting up the C2 server's security credentials. Unlike devices (which can request registration and generate an identity by simply visiting the key exchange service's URL), the registration process for servers requires the generation of a server token file (see Section 6.4). This along with the server identity must be specified in a manually generated configuration file.

The reference implementation of a C2 server also supports human and machine moderated joining. It permits the use of colour or monochrome pictograms (Section 7.3.2), word lists (Section 7.3.3), and hexadecimal notation (Section 7.3.1). It also includes a simple example of visualizing data from a device by plotting a graph, although for a real-world (production) system it would be highly desirable to utilize scalable, off-the-shelf data storage and visualization tools for time-series data — such as InfluxDb [307] and Grafana [308].

The main body of the implementation consists of less than 500 lines of Python code (plus small amounts of additional code for tasks such as generating the pictograms).

It should be noted that the C2 system implemented does not utilize any type of client-side user authentication. This was deliberately omitted from the example system for demonstration purposes. However the implementation of user authentication for web applications is a solved problem, and a number of off-the-shelf OSS solutions compatible with the Python implementation (such as Flask-Login[7]) exist and could be easily added for any real-world use.

9.6 Backend services

A reference implementation for the backend services for SRUP was also produced. This consists of two main parts: the key exchange server; and the supporting infrastructure for hosting and MQTT message brokering.

9.6.1 Key exchange server

The key exchange server was implemented as a relatively simple Python application. It implements a REST API endpoint for all of the stages of the device registration process, and again uses the Flask library to provide the web services framework. All of the data

[7] https://flask-login.readthedocs.io/en/latest/

(such as the device identities and keys) are stored locally using an SQLite database [309], however for a larger-scale full production system this could be easily swapped out for any other SQL database system (such as PostgreSQL [310]).

Table 9.1 depicts the REST end-points which are implemented.

End-Point	Type	Service
`../register/status`	GET	Returns the status of the KeyEx server
`../register/register`	POST	Performs the registration of a device ID
`../register/validate`	POST	Performs the mutual validation of exchanged keys
`../register/access`	POST	Process the device's CSR and return the MQTT access key and certificate for the device
`../register/get_key`	GET	Returns the public key for a supplied device ID
`../register/get_type`	GET	Returns the device type for a device ID
`../register/C2_check`	GET	Is C2 server registered with the KeyEx service?
`../register/C2`	POST	Performs C2 server registration with the service

TABLE 9.1: The REST API end-points implemented by the SRUP key-exchange service

9.6.2 Containerization

Although the key exchange server is relatively simple (especially when running locally), the configuration required to host this securely on a remote web-server (including the provision of a TLS certificate for the hosting domain) and the configuration required by the MQTT broker represent additional steps. Although these do not represent anything that an experienced backend web developer wouldn't be familiar with — in keeping with the goal to make SRUP as easy to use as possible, an implementation of the backend as a series of Docker containers has also been provided.

By making use of Docker [311] and the Docker-compose orchestration system [312] it is easily (and reproducibly) possible to specify a standing configuration for these components.

The main configuration for the orchestration is specified using the YAML (*YAML Ain't Markup Language*) format [313], and is contained in the `docker-compose.yml` file. This defines four micro-services — each of which is implemented as a docker container. These are described in Table 9.2.

Of these, only the *KeyEx* service is a bespoke container. The other three all use extant container images, and augment these with specific configuration settings specified in the *docker compose* file. These are presented to the containers via the filesystem, using Docker *Volumes*.

The Dockerfile specifying build process for the KeyEx image, simply takes the latest Python 3 container image, and installs the Python library dependencies for KeyEx. These include

Service Name	Container Image	Service description
`web`	`nginx:latest`	Implements nginx as a web server and reverse proxy
`keyex`	*Bespoke*	Implements the SRUP Key Service (building from the specified Dockerfile)
`broker`	`eclipse-mosquitto:latest`	Implements the Eclipse Mosquitto MQTT Broker
`certbot`	`certbot/certbot`	Implements the Let's Encrypt certbot for TLS certificate generation

TABLE 9.2: The micro-services implementing the SRUP backend

Flask, the *Cryptography*[8] library, and *Green Unicorn*[9] which implements the Python Web-Server Gateway Interface (WSGI) [314]. The final step is to specify the startup script for the KeyEx service.

Thus the entire backend may be distributed as a series of small source code files, which can be assembled by Docker and Docker-Compose to deploy a complete SRUP universe on any server with the minimal of developer intervention. Although Docker-Compose has been adopted for the reference implementation, alternative orchestration layers such as Kubernetes [315] may be substituted, especially for running large-scale deployments under high or variable load.

An architecture diagram depicting how all of the components fit together is shown in Figure 9.4.

9.7 Bootstrapping SRUP and the Key Generation Tool

The final part required to ensure that a SRUP universe can be stood up from scratch is a means to generate the set of certificates and keys that are required. For a universe running

[8]https://cryptography.io
[9]https://gunicorn.org

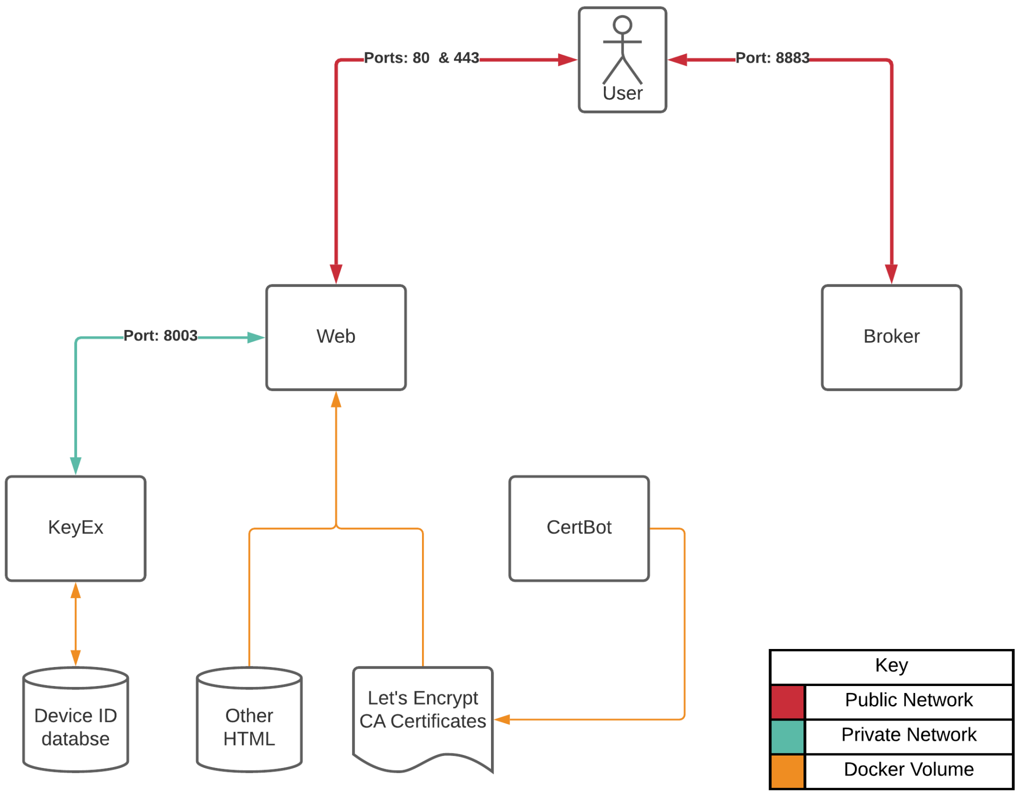


FIGURE 9.4: An architecture diagram showing the components of the containerized SRUP backend.

on an Internet connected network, a total of eight such files are necessary:

- Broker CA certificate
- C2 server public & private key-pair (used for the C2 server identity)
- C2 server token file
- Key Exchange server public & private key-pair (used for the KeyEx server identity)
- C2 broker access key and certificate (used for the C2 server's access to the MQTT broker)

For a system running on a private (or otherwise non-Internet connected network), two additional files are also required:

- Private web CA root key
- Private web CA root certificate

The process to generate these files, especially the root CA files is, whilst not especially difficult [316], fairly involved. As such, a key generation tool (also written in Python) has been created to be distributed with the other elements of the backend.

This tool is designed to generate both the initial bootstrapping configuration for a new installation of a SRUP backend universe (as described in Section 9.6.2), and to generate the necessary credentials (including the server token file) for any C2 servers which are to be used within that universe.

Figure 9.5 depicts the tool in operation, and the interrelationships between the various files generated.

This containerized approach to the deployment of a backend system also supports the intent of Research Question RQ1. By utilizing Docker, the containerized approach establishes the backend as a component which can easily be deployed by a user without needing to implement the complexities of securing the broker and managing the key exchange themselves.

This approach therefore makes it possible to consider the deployment of a SRUP universe as a commodity service. Either offered as a turn-key commodity deployment for non-technical users, or as a part of a hosting package. For individuals or organizations with a particular requirement for privacy or security of their IoT systems, they could provision the requisite backend systems for their IoT devices within their own controlled network infrastructure or hosting environments, and thus be able to provide guarantees as to the availability and accessibility of TLS keys used for MQTT traffic encryption.

9.8 Hardware

Whilst not directly addressing RQ5, the final part of the implementation work for SRUP was to design and construct a number of exemplar IoT devices to be controlled by the protocol. In addition to the device depicted in Figure 9.3 (which was used during the development of the software), two other hardware device designs were constructed and built in small numbers. These are described in more detail in the following Sections.

9.8.1 Timing device

For the purposes of the performance evaluation of the SRUP protocol (see Chapter 11) a simple IoT device was built, which consisted of a small custom circuit board, built using a prototyping board. The purpose of this board was to provide a device with a visual output (via the LEDs) and permitting user input (via a push-button) to allow the user to signal the device was ready to start the experiment.

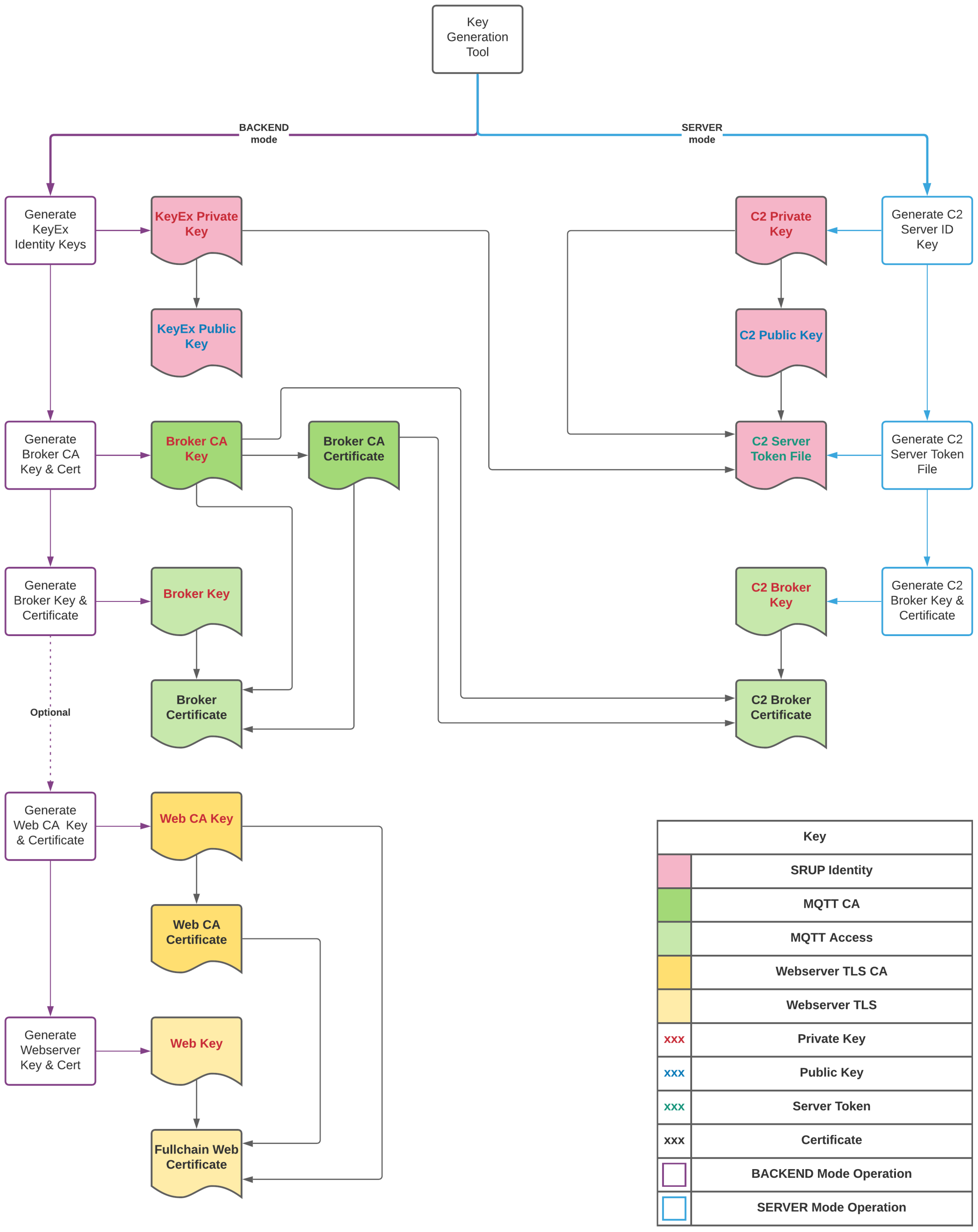


FIGURE 9.5: A diagram depicting the SRUP Key Generation Tool in operation to bootstrap a new SRUP universe, and showing the interrelationships between the various files generated.

A circuit schematic for this board can be seen in Figure 9.6, and a photograph is shown in Figure 9.7.

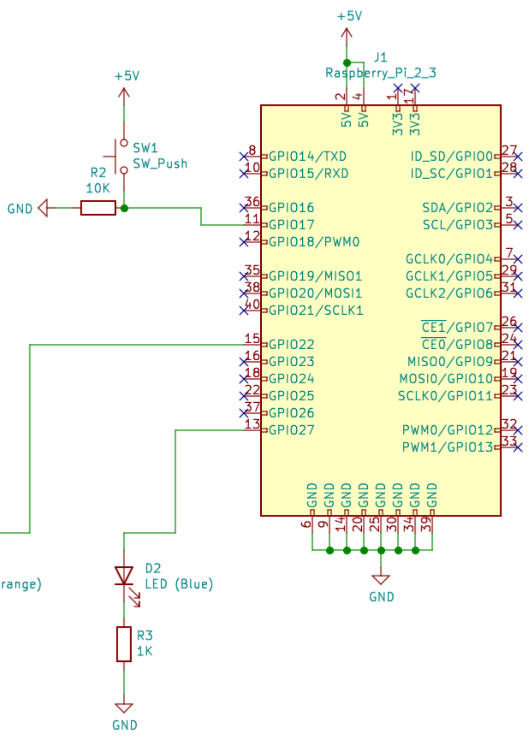


FIGURE 9.6: The circuit schematic for the simple IoT device used for the SRUP performance evaluation experiments.

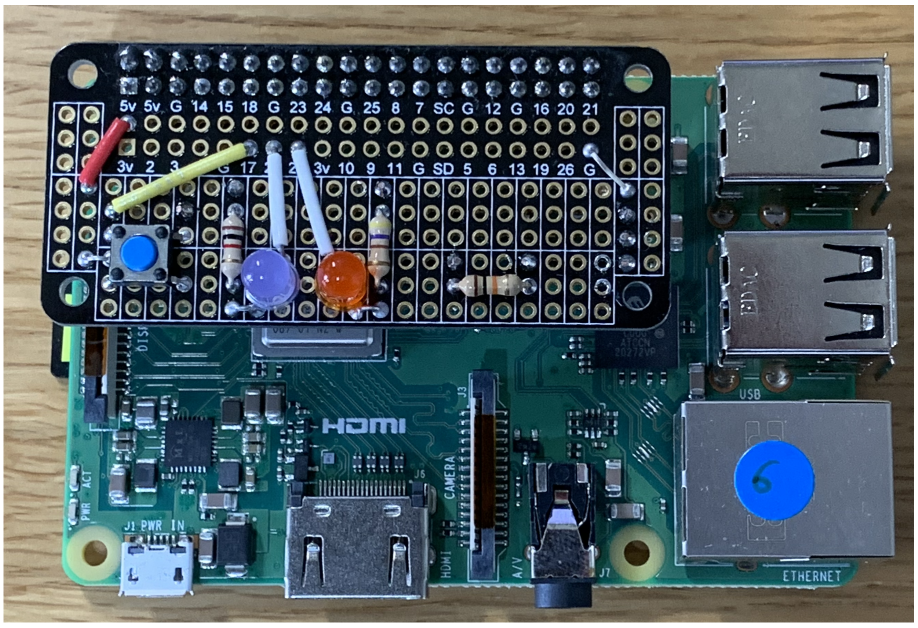


FIGURE 9.7: A photograph showing the prototyping board used to make the simple IoT device used for the SRUP performance evaluation experiments.

9.8.2 Syndication experiment device

A second device was built for the *capstone* demonstration of the research. This device was more sophisticated than the devices described previously, and incorporated two small ambient condition sensors. The sensors (connected via an I^2C connection) were each capable of measuring temperature, humidity and barometric pressure. The boards also incorporated a 3.2" colour LCD touchscreen, connected via the Serial Peripheral Interface (SPI) bus of the Raspberry Pi.

This device was designed to provide representative, real-world, data acquisition — and the attached LCD display both represented the sort of display that a room thermostat would likely be fitted with, and enabled observed C2 joining operations. The board also featured an I^2C EEPROM module, which enable it to store Linux kernel configuration settings, as specified in the Hardware Attached on Top (HAT) specification [317] (although mechanically the board did not incorporate the necessary cutouts to fully adhere to the HAT standard). The design also incorporated a number of additional features, which were not used in the eventual experiment: such as an additional (user-accessible) EEPROM module, and an 8-byte ROM hardware serial number.

Due to the relative complexity of the board design (in comparison with previous boards), this board was designed to be manufactured as a Printed Circuit Board (PCB). Using a double-sided PCB layout, and surface-mount components enabled a greater density of components than would otherwise have been possible, as well as enabling the integration of commercially available sensors.

The circuit schematic is shown in Figure 9.8.

The PCB layout for this board is shown in Figure 9.9, and a rendering of the PCB is shown in Figure 9.10. (The assembled hardware is shown in Figure 10.6).

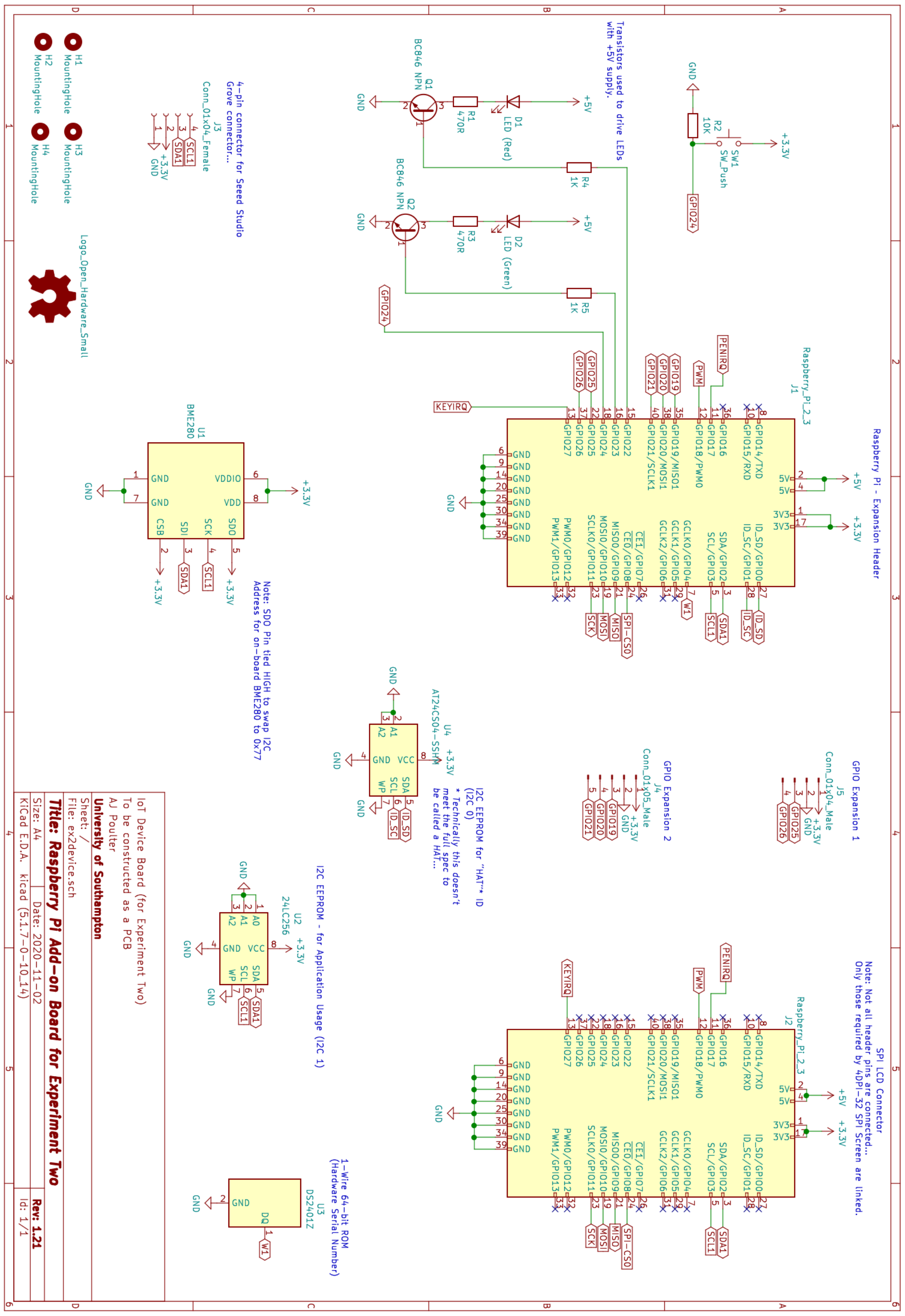

Title: Raspberry PI Add-on Board for Experiment Two
IoT Device Board (for Experiment Two)
To be constructed as a PCB
AJ Poulter
University of Southampton
Sheet: /
File: ex2device.sch
Size: A4
Date: 2020-11-02
Rev: 1.21
KiCad E.D.A. kicad (5.1.7-0-10_14)
Id: 1/1
Raspberry Pi - Expansion Header
SPI LCD Connector
Note: Not all header pins are connected...
Only those required by 4DPI-32 SPI Screen are linked.
Transistors used to drive LEDs with +5V supply.
Note: SDO Pin tied HIGH to swap I2C Address for on-board BME280 to 0x77
I2C EEPROM for "HAT"* ID (I2C 0)
* Technically this doesn't meet the full spec to be called a HAT...
I2C EEPROM - for Application Usage (I2C 1)
1-Wire 64-bit ROM (Hardware Serial Number)
4-pin connector for Seeed Studio Grove connector...
GPIO Expansion 1
GPIO Expansion 2
Logo_Open_Hardware_Small

FIGURE 9.8: The circuit schematic for the syndication experiment device

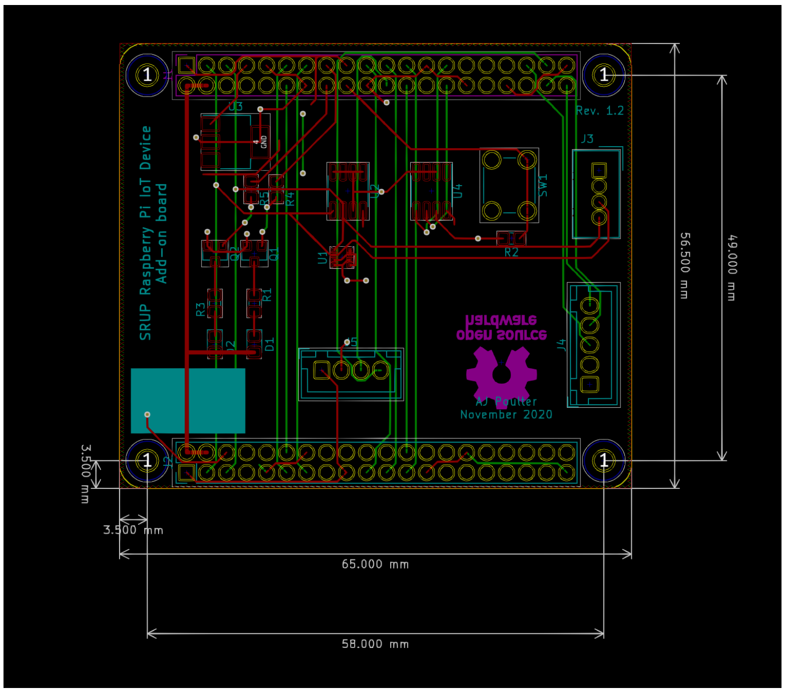


FIGURE 9.9: The PCB layout for the syndication experiment device

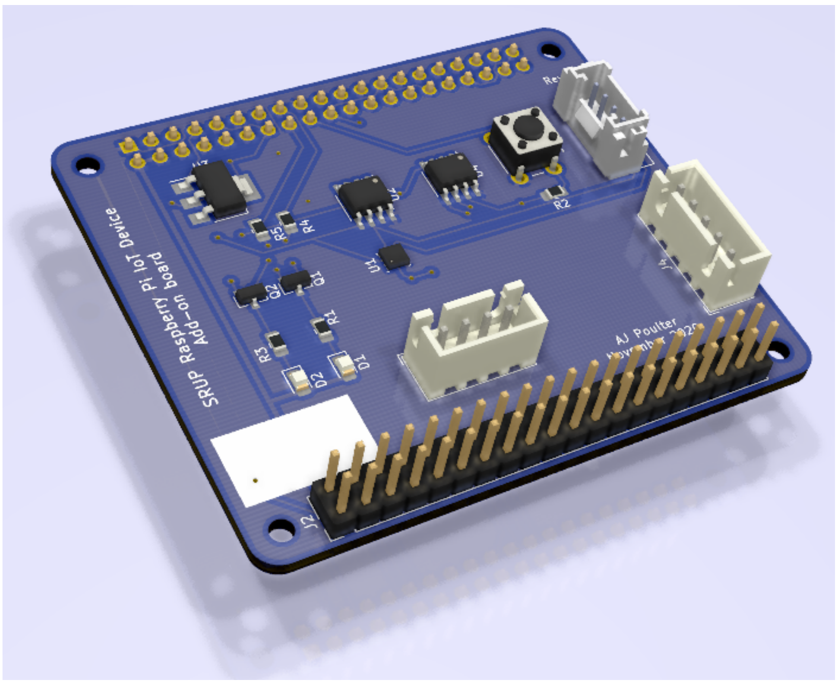


FIGURE 9.10: A rendering of the PCB layout for the syndication experiment device

The full Bill of Material (BOM) (excluding the Raspberry Pi, and passive components) is shown in Table 9.3.

Part Number	Qtty	Component	Interface	Data Sheet
DS2401Z	1	64bit ROM Memory	1-wire	https://bit.ly/33WU6qN
24LC256	2	256kbit Serial EEPROM	I2C	https://bit.ly/3tVbnLC
BME280	2	Temperature Sensor	I2C	https://bit.ly/3wicPct
4DPI-32	1	3.2" colour LCD screen	SPI	https://bit.ly/3bCiMcz

TABLE 9.3: The main Bill of Material for the Syndication Experiment Device, excluding the Raspberry Pi single board computer

This device and the capstone demonstration itself are described further in Chapter 10.

9.8.3 Other hardware

Several other single-purpose devices were built to evaluate specific aspects of the research. These included a device consisting of an eInk display fitted to a Raspberry Pi Zero W used for evaluating the word list comparison (see Section 7.3.3 and Figure 7.11), a device consisting of a colour LCD display fitted to a Raspberry Pi 3B+ for the pictographic comparison (see Section 7.3.2 and Figure 7.8), and an NFC reader to evaluate the use of RFID techniques for observation (See Section 7.4.2 and Figure 7.16).

9.9 Summary

This Chapter has described the implementation of a software library for SRUP and has demonstrated its ease of use for developing IoT systems, in comparison to hand-building an insecure solution using only MQTT. It has also described the containerized backend solution to deploy a SRUP universe, and tools to support bootstrapping such a new deployment. Example hardware devices to exploit SRUP for real-world applications were also described. In the next Chapter, the final original element of the research is described: introducing the concept of syndication, and describing how it can be applied to the problem of how to control the sharing of information between IoT C2 systems.

Chapter 10

Syndication

This Chapter introduces the concept of *syndication* to SRUP as a means to address Research Question RQ6. Specifically this aspect of the research was motivated by the requirement to be able to share both data from, and control of, devices from one C2 network to another — without requiring a trusted relationship between the two C2 networks.

Elements of this Chapter have been previously published as [7].

10.1 Sharing data and control without a fully trusted-relationship

Although the concept of C2 for the IoT is a powerful one, there are use-cases for deployed IoT devices which do not fully map onto a simple C2. An example of this is a scenario where the operators of a sensor system wish to share some parts of the data collected by those sensors with another party, but without giving that third-party full access to their devices. This is a scenario that is quite likely in the context of an IoBT system, where the military operators may wish to share some aspects of the data with non-combatant parties (such as NGOs or local law-enforcement).

As an example, consider a case where there is a deployed IoT system consisting of a number of fixed sensor devices, and where the devices belong to a controlling C2 system, all operating as a part of a smart city [318]. In the event of an emergency scenario, these devices could be augmented by combining them with a second set of mobile devices, operated by local fire and rescue services. SRUP supports the dynamic addition of devices to a C2 network, as well as commands requesting that devices transfer their registration to another C2 server (see Section 5.6.5.1), providing the C2 systems in question are operating within the same SRUP universe (see Section 6.2.2).

Figure 10.1 depicts three discrete C2 systems that may be operating within the same location, and which in some circumstances may wish to be combined by their operators.

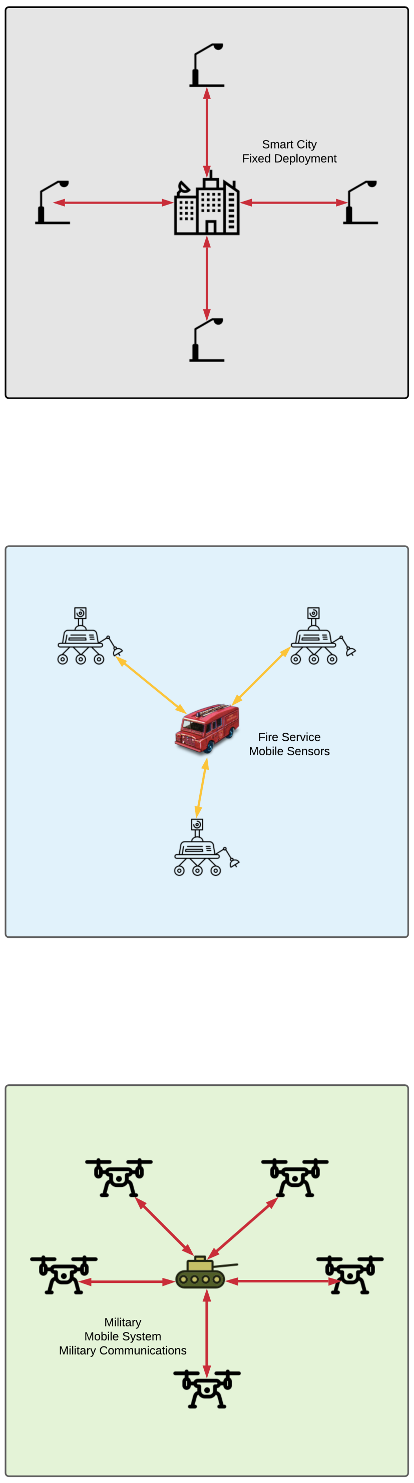


FIGURE 10.1: An example of three IoT-based sensor systems, which may be required to operate together in an emergency scenario.

For the purposes of this Chapter, consider a hypothetical use-case of a network of fixed sensors, installed to measure air quality and to provide real-time alerting to citizens if the air quality deteriorates below a certain threshold. This sensor network is owned and operated by a metropolitan region, such as a city.

In the event of a major fire or other disaster, it may be desirable for this system to be augmented with a number of additional mobile sensors, such as may be deployed by the local fire brigade. Within SRUP this can be easily achieved by the fire brigade's sensors being commanded to join the city's C2 network. This *combined* C2 system is depicted in Figure 10.2.

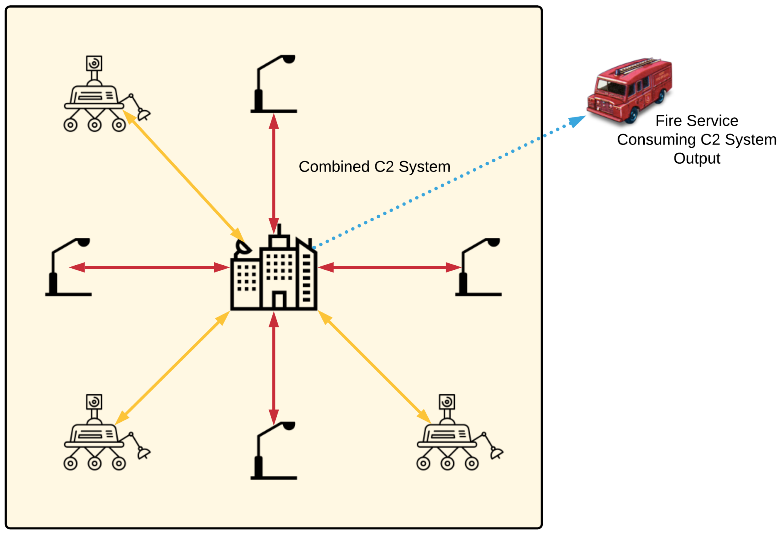


FIGURE 10.2: An example of a mobile system joining its sensors to a fixed system during an emergency scenario.

In order to create this combined C2 system, it is necessary to add the fire service's mobile sensors to the city's fixed sensor network, and to then add the fire service to the city's C2 system as a user. More generally, for this to be possible, it is asserted that the following three conditions must all be true:

1. The systems must use (or be compatible with and be able to use) the same SRUP universe
2. The owner of the sensors which are to be joined to the existing C2 system accepts (temporarily) losing ownership of the sensors to the operator of the fixed system
3. The owner of the fixed system accepts the new sensors, and takes temporary responsibility for them

Conditions 2 and 3 are especially important. Since the devices in question will become full members of the new C2 network, both parties must be willing for the systems to be joined, implying a degree of trust. For example, the side hosting the C2 network needs to trust that the devices joining are not compromised with malware, and the side providing the devices needs to trust that the other party will be diligent to ensure that they do not become so, and that they will be returned to their control at the completion of the operation.

There are scenarios, however, where such a join is not possible to achieve because one or more of the preconditions are not, and cannot, be satisfied (for example, when one party is unwilling to assign their devices to the C2 network operator's control — or where the devices in question necessarily utilize different backend systems).

Combining C2 systems is not always required however. Depending on the specific requirements, it is also possible to simply provide access for the third-party into the existing C2 system, or provide simple data export (either in the form of static data, or in the form of a live API).

10.2 Syndication concept

To explore this further, imagine that the emergency scenario described in Section 10.1 is sufficiently serious that, in addition to resources from civilian fire and rescue services, other specialist resources are brought in to provide assistance to the regular civil authorities. For example, a situation such as may occur in the event of a natural disaster. This additional assistance may be provided by military units, NGOs, or even specialist commercial providers.

In this example, assume that neither condition 1 nor condition 2 hold true; the sensors in question use their own backend system (and potentially a different cryptographic standard too, especially in the case of a military system) and the provider of the sensors wishes to retain full control over their devices. Additionally, in this scenario the provider is not willing to share *all* of the data with the city, but they do wish to provide a live feed of some of their data to the city's C2 system. Reluctance to share a totality of the data may be due to commercial, security, or other reasons. Similarly other data may not be relevant to the situation for which the systems are cooperating, and so may not be shared to avoid overloading or confusing operators with unnecessary information.

To address this requirement, the concept of *Syndication* has been developed. Syndication is a term coined by analogy to newspapers, where syndicated content may be reprinted in other publications where a pre-existing relationship exists to the original publisher. Within SRUP, syndication provides a mechanism for two C2 systems — operating within different universes — to collaboratively share data and commands between them.

Syndication not only permits sharing of data without ownership of devices changing, it also enables the C2 server corresponding to the source of the information to moderate the information flow, thereby controlling which data types (and data from which platforms) are shared.

By the addition of a specialized *syndication device* to act as a bridge between the two universes, it is possible to facilitate communications between C2 systems that do not have a cryptographic algorithm in common.

Although syndication is primarily intended to link networks which operate in different SRUP universes, the technique could also be applied to discrete C2 networks operating within the same SRUP universe.

Given the increase in utilization of IoT devices, the need to facilitate sharing of data between discrete C2 networks is important in order to enable dynamic cooperation between the operators of deployed services, especially in the context of future smart cities where standing networks of sensors or other devices may need augmentation with additional capabilities provided by third-parties during times of emergency or crisis.

10.3 Syndication messages

In each of the following Sections, the setup is assumed to be as follows:

- There are two backend *universes*: Universe A and Universe B.
- There is a SRUP C2 server which controls one or more devices and which is willing to share data with another C2 server. This is the *syndicated server* and it operates within Universe A.
- There is a second C2 server, which operates within Universe B. This server is the *syndicating server*, the server which wishes to receive data from the syndicated server.
- There is also a *syndication device*. This device is subordinate to the syndicating server, primarily operating within Universe B. It only makes connection to Universe A during syndication operations, when it forms a bridge between the two universes.

A sequence diagram illustrating the data flow during syndication operations can be seen in Figure 10.3.

It is also assumed that the ability to initiate a syndication between two C2 systems will be controlled by a pre-shared secret. This may be a static secret, or dynamically generated. This is, however, an implementation detail, and in either case the operators of the C2 systems would need to communicate this secret *out-of-band* to the SRUP protocol. Use of a pre-shared secret is not mandatory, and insecure C2 servers could also be configured to permit syndication requests without validation if required.

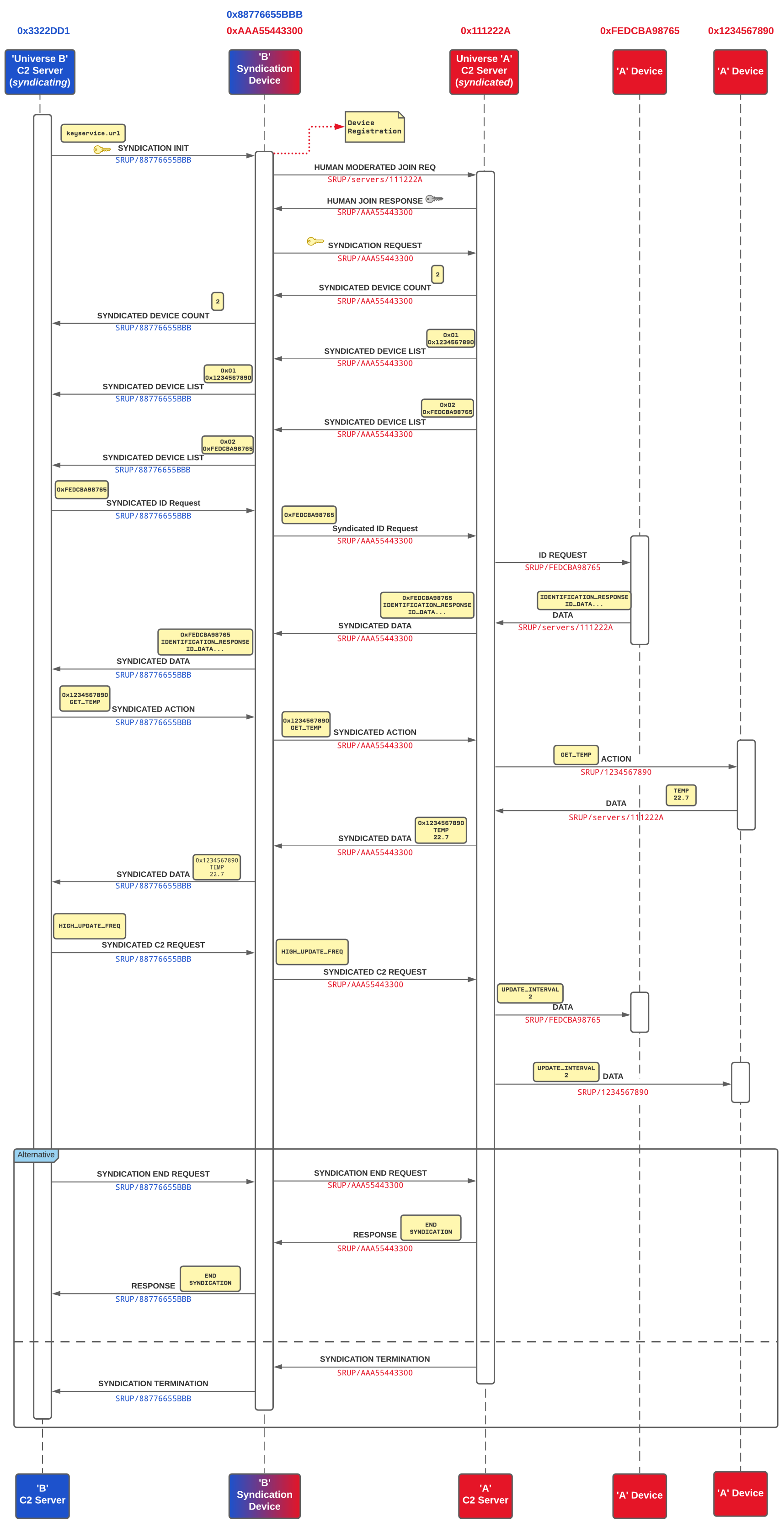


FIGURE 10.3: A sequence diagram showing an example data exchange between the devices during SRUP syndication operations. The text beneath the message-type line denotes the MQTT topic used by SRUP to carry the message

10.3.1 Syndication initialization

To begin syndication operations, it is first necessary to send a *syndication initialization* message from the syndicating C2 server to the syndication device in order to initialize the join. This is sent using the Universe B backend, keys and cryptographic protocols, via the syndication device's MQTT topic.

This message contains the URL for the Universe A key service registration, and the pre-shared secret to be used to authenticate the syndication.

On receipt of this, the syndication device should attempt to connect to the provided URL and perform the initial registration process with a new device identity to be used within Universe A. It should perform the subsequent MQTT operations using a second MQTT client connection (and using the necessary keys and certificates received as a part of the registration to the new universe).

10.3.2 Syndication request

Having successfully joined the universe A C2 system (likely requiring a human or machine moderated join), the syndication device will then send a *syndication request* message. It is sent using the syndication device's Universe A identity for the MQTT topic, and includes the pre-shared secret that the syndication device received as a part of the syndication initialization message.

10.3.3 Syndicated device count and syndicated device list

The process for determining the validity of the syndication request is application-specific, and is not enforced by the protocol. However, having received a valid syndication request message, and having determined that the request is acceptable, a C2 server which is to become syndicated must send a *syndicated device count* message to the syndicating C2 server, stating the number of subordinate devices that it intends to syndicate. This is initially sent to the syndication device (using its Universe A identity), and then relayed by the syndication device, to the syndicating server (using its Universe B identity and topic when communicating with the syndicating server). This approach of message relaying by the syndication device — using its two identities — is similarly adopted for all other syndication message types.

After a suitable message propagation delay, the syndicated server begins sending *syndicated device list* messages. These consist of two fields: the identity of one of the devices to be syndicated, and an integer index value to indicate which of the devices in the list of n devices that identity pertains to (where n is the value sent in the syndicated device count message). In all cases, the identity that is sent is the Universe A identity of the end device, since devices that are syndicated (by definition) do not have any identity within Universe B.

10.3.4 Syndicated ID request

After the receipt of a syndicated device list message, the syndicating server may optionally send a *syndicated ID request* message to the syndicated server, asking that the ID request string corresponding to the device in question is sent to it. The ID data may either be sent directly by the syndicated server, via a *syndicated data message* containing the ID request data it has previously received from the device, or may be passed on to the end device (in the form of a *regular* ID request message) to enable the device to update this information. This determination is made by the configuration of the syndicated server.

10.3.5 Syndicated data

Syndicated data messages are used by a syndicated C2 server to send data on behalf of one of its subordinate devices. These may be as a part of an ID Request, or as a part of sending operational data from a subordinate device. In either case, the message consists of three elements, over and above the standard SRUP message:

- the source ID — containing the ID of the device from which the data originates
- the data ID — used identically to the non-syndicated SRUP data message to indicate the meaning of the data
- the data value itself

Syndicated data messages always originate from a syndicated C2 server, and never directly from an end device. This means that the syndicated C2 server always has the ability to moderate what is sent. In practice, this means that a syndicated C2 server can easily be configured to share only certain types of data (e.g. temperature and humidity, but not barometric pressure), to share data only from certain devices, or to *downgrade* the precision of data before it is sent. For example, a military operator may elect to share a reduced-precision form of geolocation data with civilian authorities.

10.3.6 Syndicated action

In addition to receiving data from a syndicated C2 server, a syndicating server may make requests for actions to be performed by the syndicated server's devices. As with the data messages, there is no direct connection between the syndicating server and the syndicated devices, so all requests are moderated by the syndicated C2 server. The syndicated C2 system can therefore be configured to permit or deny different action types, for different devices, or simply accept any requests and pass them on to the corresponding target device.

The *syndicated action message* consists of the target device's identity, as well as the 8-bit integer value corresponding to the action type that is being requested.

10.3.7 Syndicated C2 request

The *syndicated C2 request message* allows for the syndicating C2 server to send a request (and any associated data) to the syndicated C2 server. This message consists of an 8-bit integer C2 request ID (corresponding to one of up to 256 pre-defined C2 request types), along with a byte-stream containing any data that may be required to support this request type.

This can be used, for example, to request to change a parameter of the server's operation such as the update frequency of the sensors (which would not necessarily require any supporting data), or for a more complex request such as requesting that the syndicated server applies a software update to its devices (in which case the details would be supplied within the data field).

In common with all of the other syndication messages, the syndicated server must determine whether or not to accept the request. The specifics of how it does this are, of course, implementation specific: but this could be an automated decision, or it could be passed to the operators of the syndicated C2 server to make the decision as to whether or not to accept the request.

10.3.8 Syndication termination and syndication end

Either party within the syndication can end the syndication session at any time. There are two ways that this can be achieved:

- The *syndicating* server may send a *syndication end request message*. This is used to inform the syndicated server that it should stop sending syndication messages, and to instruct the syndication device to stop accepting messages from the syndicated server once it has received a response message with a status of `END SYNDICATION`.
- The *syndicated* server may send a *syndication termination message*, which is used to inform the syndicating server that the syndicated server will no longer send syndication messages.

These represent a situation analogous to either party *hanging up* during a telephone conversation: either the caller signalling that they have finished the conversation, or the recipient of the call electing to end it.

There is no mechanism within SRUP for any other party to end active syndication.

10.4 Syndication example

Figure 10.4 shows an example of how syndication might work, in the context of the scenario described in Section 10.2. In this example, the military units provide a syndication device, which is compatible with both their own universe and the universe in which the civilian systems operate. The syndication device, would then be joined to the civilian C2 system, and used to initialize syndication with the military C2 system. The military C2 system can then provide a suitably moderated version of the data available to it, which can be used by users of the civilian authority's C2 system.

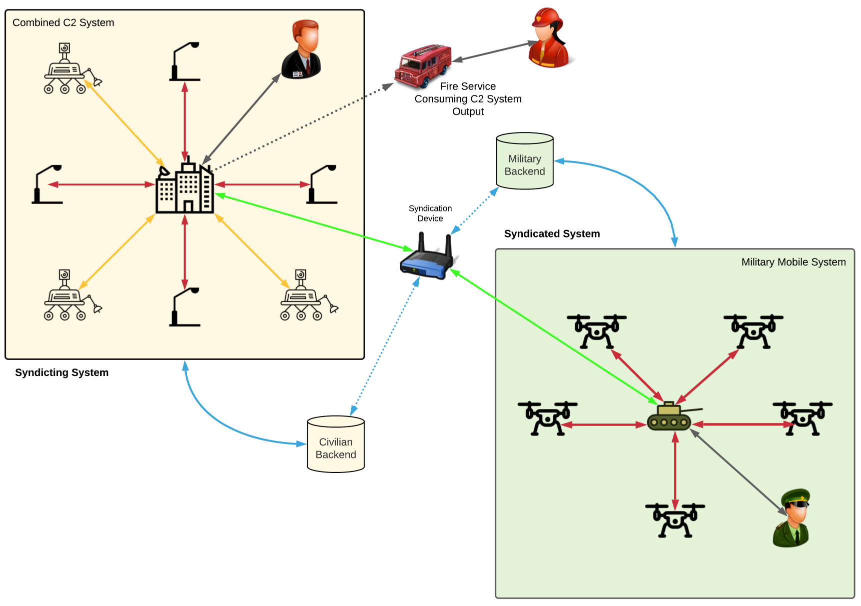


FIGURE 10.4: An example of syndication in action, showing how output from a restricted-access system can be syndicated with existing fixed, and mobile sensors.

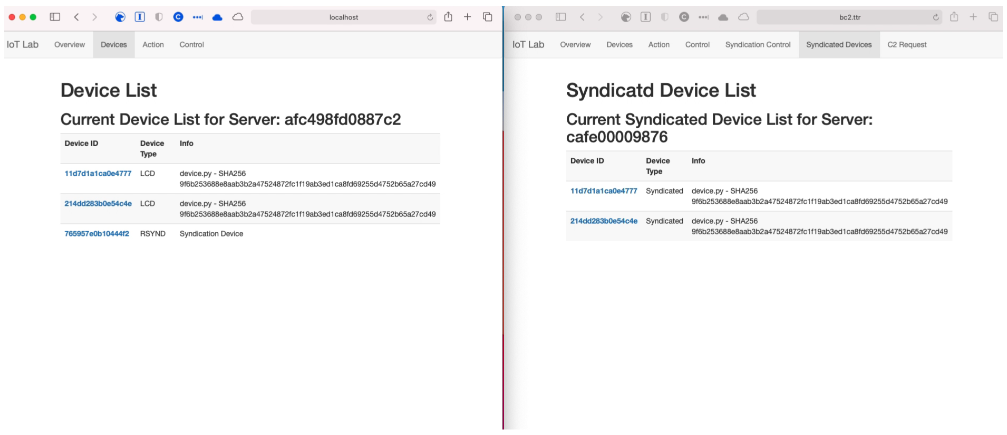
There may also be a requirement for mutual, bidirectional, data exchange between two SRUP universes. For example, sharing of the city's data with the military C2 system, as well as sharing the military data with the city. The syndication approach described here could easily be adopted by both parties. Such a mutual syndication approach would enable each to be both syndicating and syndicated: each with their own set of rules about data sharing, processing action requests, and so on. Depending on the specifics of the situation, and how the data is being used, there may be additional concerns about the trustworthiness of the data being ingested into a more secure system — although the syndication process easily allows data received this way to be segregated by the C2 system if required.

Although the initial implementation of syndication within SRUP C2 servers was built to permit just one syndication session at any time, the underlying SRUP protocol has no such restrictions and a C2 server could support simultaneous bidirectional, syndication to multiple other systems.

10.5 Experimental implementation

An implementation of the syndication approach was created for SRUP, building the additional message types onto the software stack described in Chapter 9. This implementation was then utilized to conduct a small-scale table-top experiment to establish the correct operation of the syndication process.

For the purposes of this experiment a total of twelve of the room monitoring IoT devices (see Section 9.8.2) were deployed within a laboratory setting, along with three separate instances of a web-based C2 interface (see Section 9.5). An example of two of the C2 interfaces (showing syndication in action) can be seen in Figure 10.5), and an example of one of the deployed hardware devices is shown in Figure 10.6.



Device List

Current Device List for Server: afc498fd0887c2

Device ID	Device Type	Info
11d7d1a1ca0e4777	LCD	device.py - SHA256 9f6b253688e8aab3b2a47524872fc1f19ab3ed1ca8fd69255d4752b65a27cd49
214dd283b0e54c4e	LCD	device.py - SHA256 9f6b253688e8aab3b2a47524872fc1f19ab3ed1ca8fd69255d4752b65a27cd49
765957e0b10444f2	RSYND	Syndication Device

Syndicatd Device List

Current Syndicated Device List for Server: cafe00009876

Device ID	Device Type	Info
11d7d1a1ca0e4777	Syndicated	device.py - SHA256 9f6b253688e8aab3b2a47524872fc1f19ab3ed1ca8fd69255d4752b65a27cd49
214dd283b0e54c4e	Syndicated	device.py - SHA256 9f6b253688e8aab3b2a47524872fc1f19ab3ed1ca8fd69255d4752b65a27cd49

FIGURE 10.5: A screenshot from the example implementation of the web-based C2 system during syndication operations.

Both syndicating and syndicated C2 networks were constructed, and a full set of syndication operations were demonstrated to work. For the purposes of this experiment the web-based C2 interface was configured to enable a subset of the data from devices joined to the syndicated network to be viewed in the C2 system for the syndicating network, and the syndicating system was able to make action requests to the syndicated devices.

An architecture diagram showing the full experimental setup can be seen in Figure 10.7.

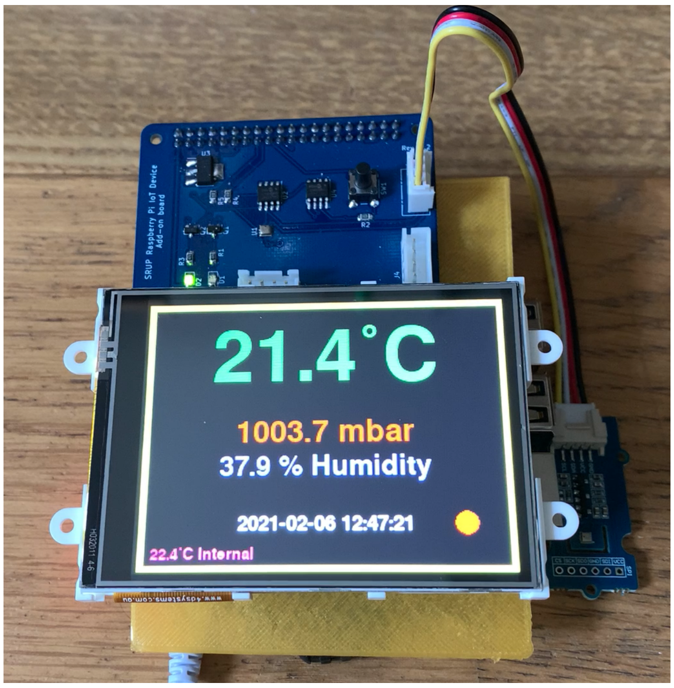


FIGURE 10.6: An example of the Raspberry Pi based hardware device used in the experimental assessment of syndication operations.

A demonstration of SRUP syndication operations can be seen at:
https://www.youtube.com/watch?v=F0_qlqh0Oiw&t=459s [319].

10.6 Guest user

Although the concept of syndication was conceived in relation to wanting to share data between discrete C2 systems, during its development it became apparent that the concept can also be used to solve the problem of how to provide guests with limited or partial access to *smart devices* in an area they are visiting.

The problem of providing guest access is not new, but as the prevalence of IoT devices increases, the degree to which they become tightly integrated with their C2 systems is likely to grow. As a consequence of increased complexity, the full range of an installed device's functionality may not be accessible without a user also having access to the associated C2 system's user interface (typically provided via a smartphone app). Although simply giving a guest access to a building's C2 system is relatively simple, the challenge is to be able to securely revoke that access once the guest has left (or is otherwise no longer permitted access).

An example of the SRUP syndication solution is depicted in Figure 10.8. Here the guest user is given time-bounded access to a subset of the devices on the owner's C2 system, in

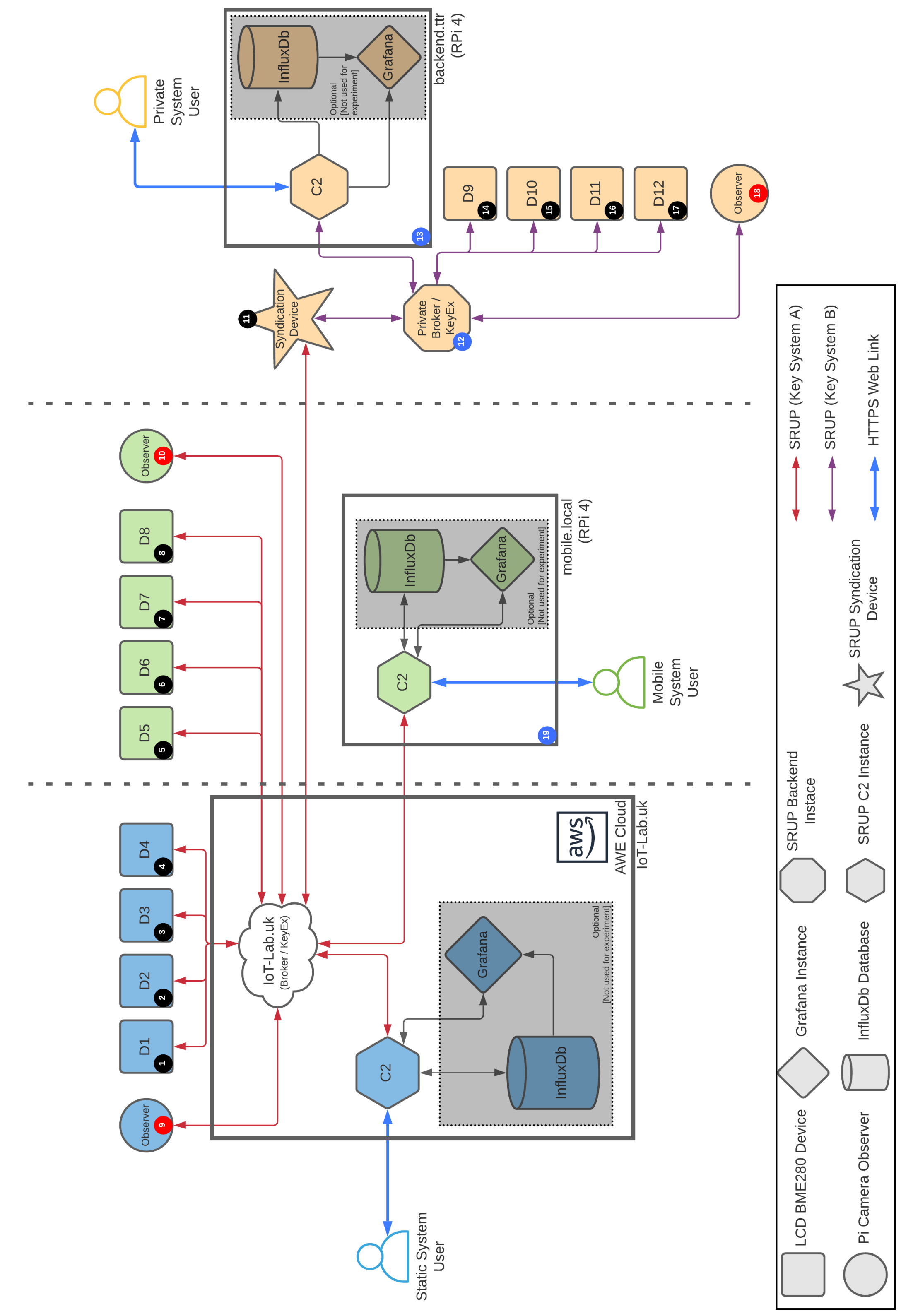


FIGURE 10.7: An architecture diagram showing the experimental setup used for a practical evaluation of syndication operations

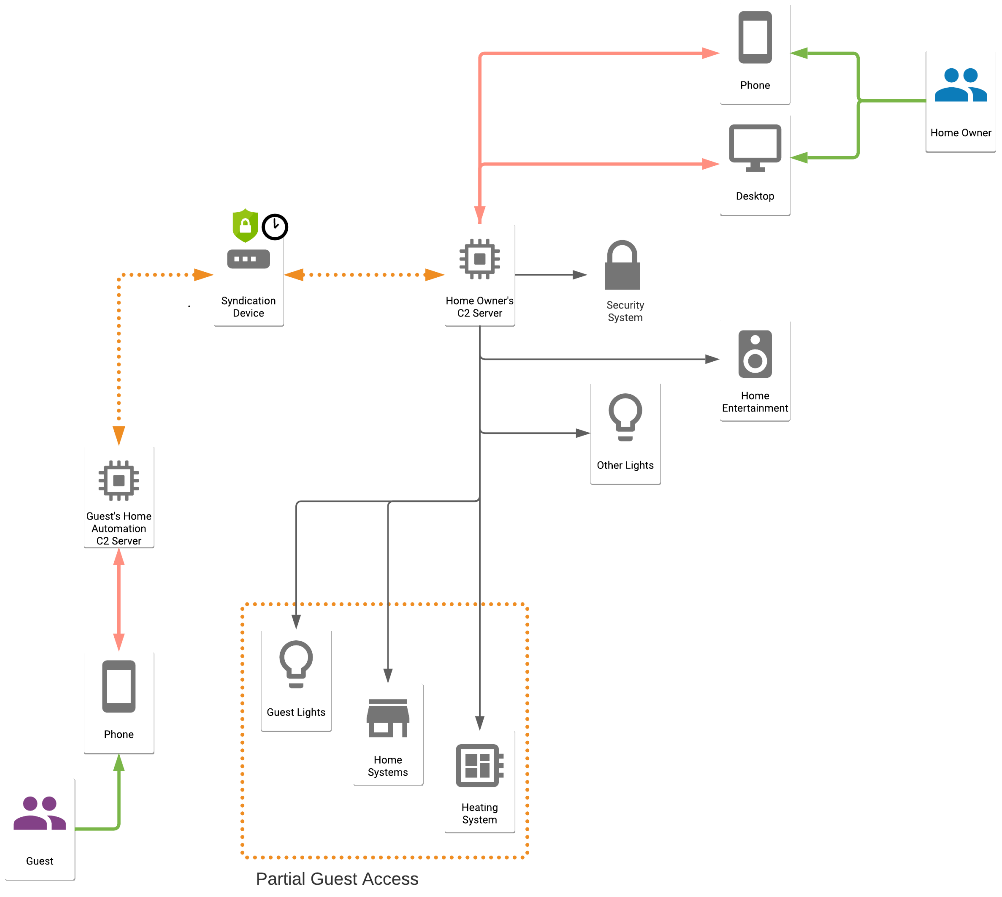


FIGURE 10.8: An illustration of the integration of guest access to a C2 system, utilizing syndication to provide moderated access to parts of the system.

order to facilitate access to only a selection of available devices, with more sensitive or secure devices being restricted to control by the system owner only.

By adopting an approach based around SRUP syndication the guest user could add access to the permitted devices to their existing C2 instance, integrating them alongside any other devices that they either own or have been granted access to, as a part of their own personal C2 console. Guest access can subsequently be securely revoked by the owning C2 system (either automatically after a designated period of time has elapsed, or at the request of the C2 system operator manually revoking access) without effecting other user's access to the devices. The use of syndication would also make it possible to dynamically moderate the data and control shared, based on criteria such as location, time of day, or the state of other devices within the system. For example an implementation of this type could limit the guest user's access to certain devices to during the daytime only, or restrict to occasions when the user is physically at the location in question. Similarly, by utilizing the state of other devices within the system, control may be granted to devices on the basis of certain conditions being met — for example, smoke detector activation could enable the unlocking of windows or doors.

Although this approach to combining C2 interfaces for domestic devices flies somewhat in the face of the current *walled garden* approach to IoT device control (where each new device typically requires its own bespoke app to permit the user to interact with it) it aligns well with the growing *hub-based* approach used to permit devices such as digital assistants to control multiple systems of devices from a single interface. Although currently this latter approach does not use a unified C2 model, and instead relies on API hooks into REST interfaces, the adoption of such an approach as described here and previously would enable superior integration, whilst still enabling more advanced functionality to be controlled from a dedicated app.

The adoption of open standards for IoT C2, alongside the provision of freely available OSS implementations of backend services, would also help to eliminate the problems caused when hardware vendors remove support for older products. If manufacturers adopted open standards, then instead of leaving customers with unusable devices when vendors cease provision of backend services, those customers would be able to provision their own backend services (either locally or via cloud-based servers) or purchase third party provision from a service provider.

10.7 Summary

This Chapter has described syndication in the context of SRUP and examined how it can be utilized to address the requirement to be able to control the sharing of information between adjacent C2 systems, as well as describing its use for the related problem of how to permit time-bounded guest access to IoT systems. The next Chapter will describe in detail the experimental work to assess the performance of the SRUP protocol in simulated real-world conditions, and examine the overhead costs associated with running the protocol, in terms of additional processing time, message size and power consumption.

Chapter 11

Experimental Assessment of the Performance of SRUP

This Chapter looks to answer the final Research Question (RQ7), by describing a set of experiments conducted to evaluate the performance of the SRUP protocol. These experiments were designed to compare the relative processing time, message size and power consumption of a device using the SRUP protocol, with an identical device taking an insecure approach, and directly utilizing *plain* MQTT messages.

Elements of this Chapter have been previously submitted for publication as [8].

11.1 Execution time analysis

In 2017, an initial assessment of the protocol's performance was made, by timing the execution of the cryptographic functions used to support the SRUP protocol. A test case was evaluated, measuring the time taken to sign and verify a representative SRUP update initiate message (selected as it is the largest of the SRUP message types). This process was run five times on a Raspberry Pi 3 B, with the mean time calculated. As per the standard implementation of SRUP the signing functions used SHA-256 to generate the message hash, and then an RSA signature was then calculated for this hash value. This signing process took an average of 56.68 ms, and the average time to verify the signature was just 9.910 ms. The full data set for this initial experiment is available [320], and further details of the testing are described in Appendix C.

11.2 SRUP and MQTT performance comparison

A more formal set of experiments were conducted in 2020. These experiments consisted of comparing two cases (SRUP messages, and simple unencrypted MQTT messages) across a number of different network conditions. In the first of these cases SRUP action messages were created, sent over a TLS encrypted connection via a broker, decoded and validated by the recipient and finally processed. In comparison, for the direct use of MQTT the process consisted of sending the message over an unencrypted MQTT topic, and was then processed by the receiver. Three sets of measurements were taken: the average time taken between sending the message and the receiver responding; the power consumption of the device; and the overall message data packet size.

The comparison between the two cases allows for the assessment of the overhead added by the use of the cryptographic algorithms. Since the direct MQTT case makes no use of either TLS for the MQTT message traffic encryption or RSA for message signing, comparing the two approaches enabled the measurement of the overhead caused by these elements.

When using SRUP, there is a one-time key exchange process which only occurs when the device initially joins the C2 network. This process results in a short additional time delay, which is not part of the usual operation of the protocol. This registration and key exchange process uses HTTPS rather than MQTT to retrieve the key. As there is no equivalent step within the MQTT-only setup, the key exchange element was deliberately excluded from the comparison experiment.

A diagram depicting factors associated with the total processing delay for a SRUP message can be seen in Figure 11.1.

11.2.1 Hardware

The experimental setup consisted of five IoT devices, each built from a Raspberry Pi 3B+ single board computer, fitted with a custom circuit board which included two LED status indicators and a push button for user interaction. An example of the device can be seen in Figure 11.2.

All of the experiments were performed in lab conditions. The devices were connected over Ethernet to a Raspberry Pi 3B+, acting as the C2 server. The C2 server was running custom software that selected one of the five devices at random, sent a message to that device requesting that it toggled the state of the LED, and then waiting for a random interval before looping back. The program execution continued until each device had received a total of 250 messages. This workflow is illustrated in Figure 11.3. Two separate

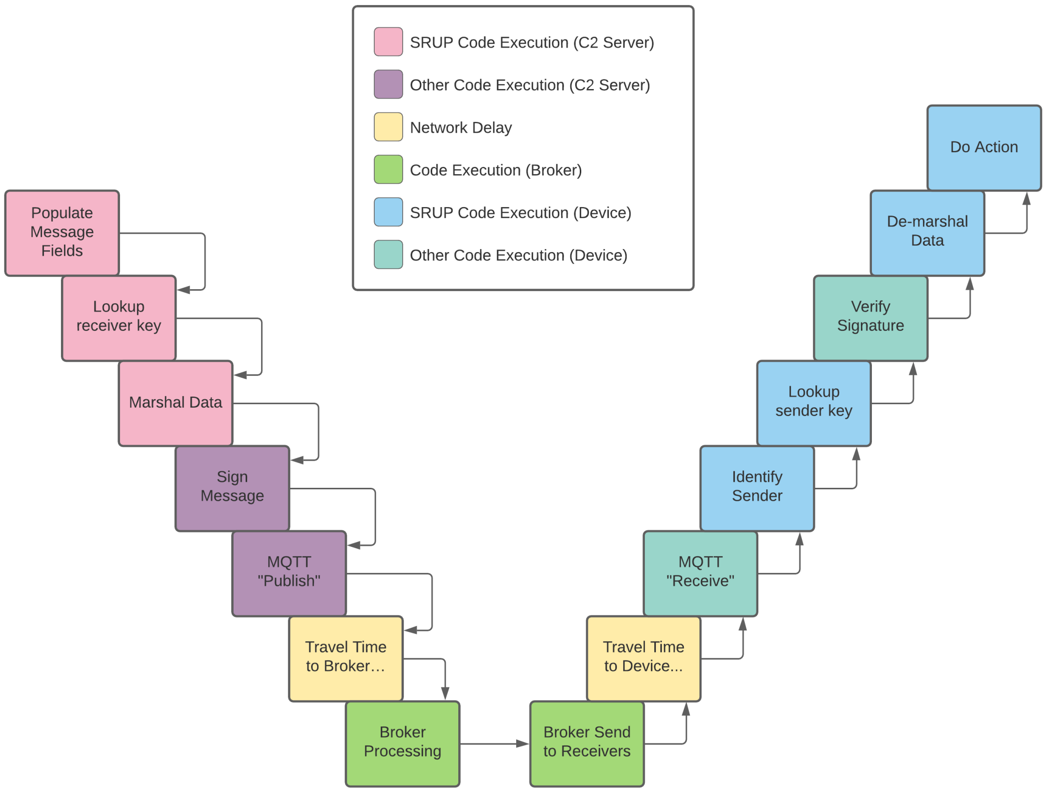


FIGURE 11.1: A diagram showing the information flow associated with the processing of a SRUP message, which contribute to the time taken.

implementations of the C2 software were written. One using SRUP action messages over the TLS encrypted MQTT connection, and one using a simple plain-text MQTT message.

The *a priori* assumption was that a significant proportion of any additional delay would be due to the time taken to process the cryptographic algorithms used for message signing. In order to evaluate the extent to which the performance of the protocol is influenced by the speed of the hardware, an additional device was built using a faster Raspberry Pi 4 single board computer, allowing for a performance comparison with the other devices. The Pi 4 device was identical to the devices described above, apart from the CPU and the RAM (Random Access Memory) configuration changes between the Pi 3 and Pi 4. A summary of the configuration of the two systems is shown in Table 11.1

Model	CPU	CPU Clock	RAM Size	SDRAM Type
Raspberry Pi 3B+	Broadcom BCM2837B0 Quad-core Cortex-A53 ARMv8 64-bit	1.4GHz	1GB	LPDDR2
Raspberry Pi 4	Broadcom BCM2711 Quad-core Cortex-A72 ARMv8 64-bit	1.5GHz	2GB	LPDDR4

TABLE 11.1: Specification differences between the Raspberry Pi 3B+ and Raspberry Pi 4 Single Board Computers used within the experiments.

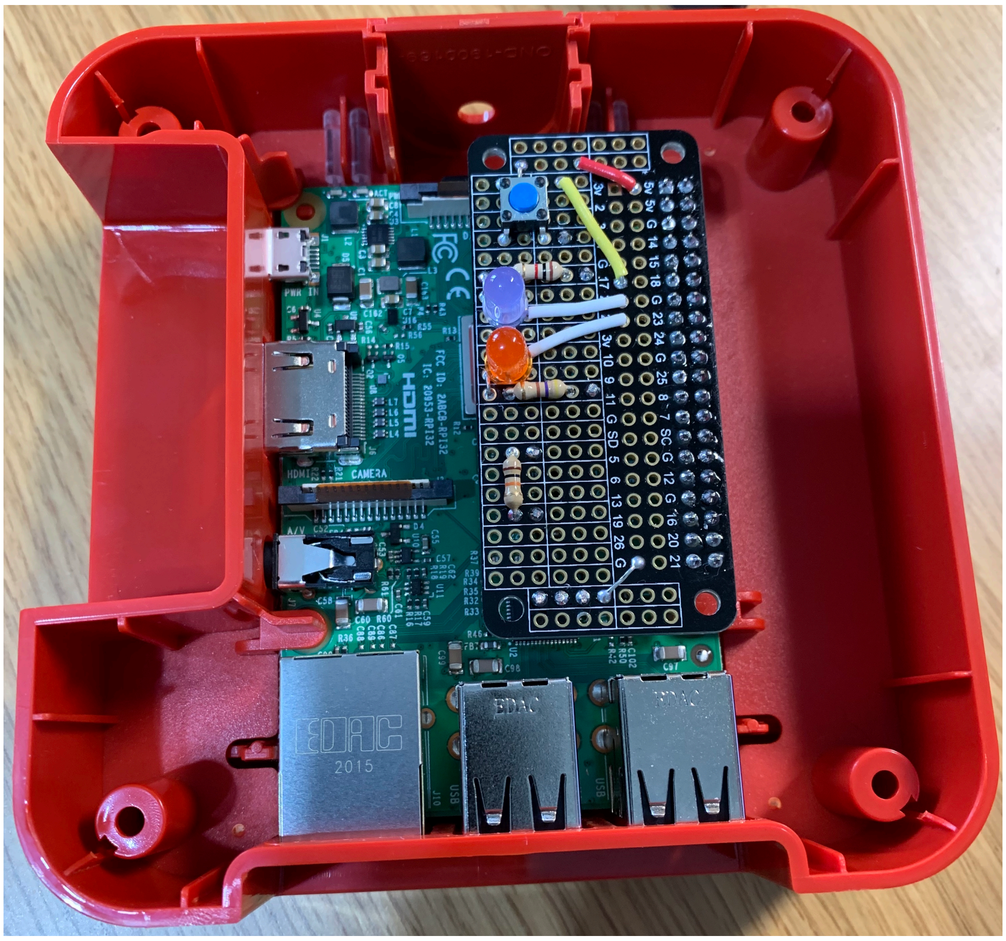


FIGURE 11.2: The experimental hardware, consisting of a Raspberry Pi 3B+ single-board computer, and a custom circuit board.

11.2.2 Software

The devices utilized the software stack described in Chapter 9, which enabled the device code to consist of a short and easy-to-understand Python script. Listings of the software can be seen in Appendix D.

11.2.3 Time synchronization

In order to measure the temporal overhead, the experiment made use of log files generated by the devices and by the C2 server. These logs indicate the time at which the server initiated the generation of the command and the time at which the receiving device had processed the message. The timestamps were subsequently used to calculate the elapsed *wall-clock* time for the operation. In order to ensure that the clocks on both devices were synchronized as accurately as possible, the device acting as a C2 server was also configured to act as an NTP [281] server using the *chrony* tool[1]. Since there was no

[1] https://chrony.tuxfamily.org

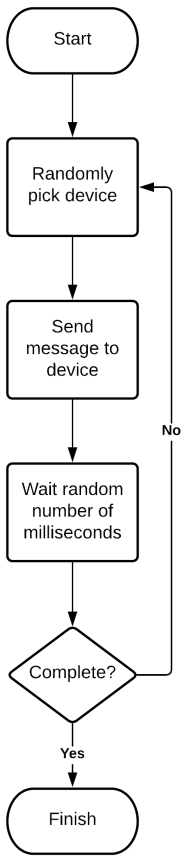


FIGURE 11.3: A flowchart showing the execution of the C2 server during the experiment

requirement for precise synchronization to an absolute time reference, the C2 server was running as the authoritative time source on the local network. Configuring the device's clocks to use the local time-server as their sole source of time ensured that the clocks were as tightly aligned as possible.

11.3 Network conditions

Due to the difficulty in deploying devices in remote locations caused by the ongoing COVID-19 (SARS-CoV-2) pandemic, combined with the greater reproducibility from using controlled conditions, all of the network conditions were simulated, rather than using real-world cellular data services.

11.3.1 Network condition simulation

For the purposes of the experiment, it was assumed that the experimental hardware represented deployed IoT devices, connected to a C2 server over a cellular data connection. Network condition simulation was used in order to appropriately represent the behaviour of a cellular network in different operating environments. This simulation was conducted using the Linux *tc* tool [321] and associated *tcconfig* wrapper tools (such as *tcset* [2]) [322].

Work by Khatouni, Trevisan, and Giordano provides performance data for cellular networks in a range of conditions [323], and this was used to select representative parameters for tc. In order to assess the performance comparison across a broad-range of conditions, ten different network conditions were selected. These were drawn from a software implementation of the Khatouni, Trevisan, and Giordano dataset produced by Trevisan [324]. The cases selected ranged from the theoretical 'best case', where all devices were running on the same Local Area Network (LAN) without additional delays, through to simulations of 4G and 3G cellular networks in 'good' to 'medium' and 'poor' signal conditions.

11.3.2 Operation in austere network conditions

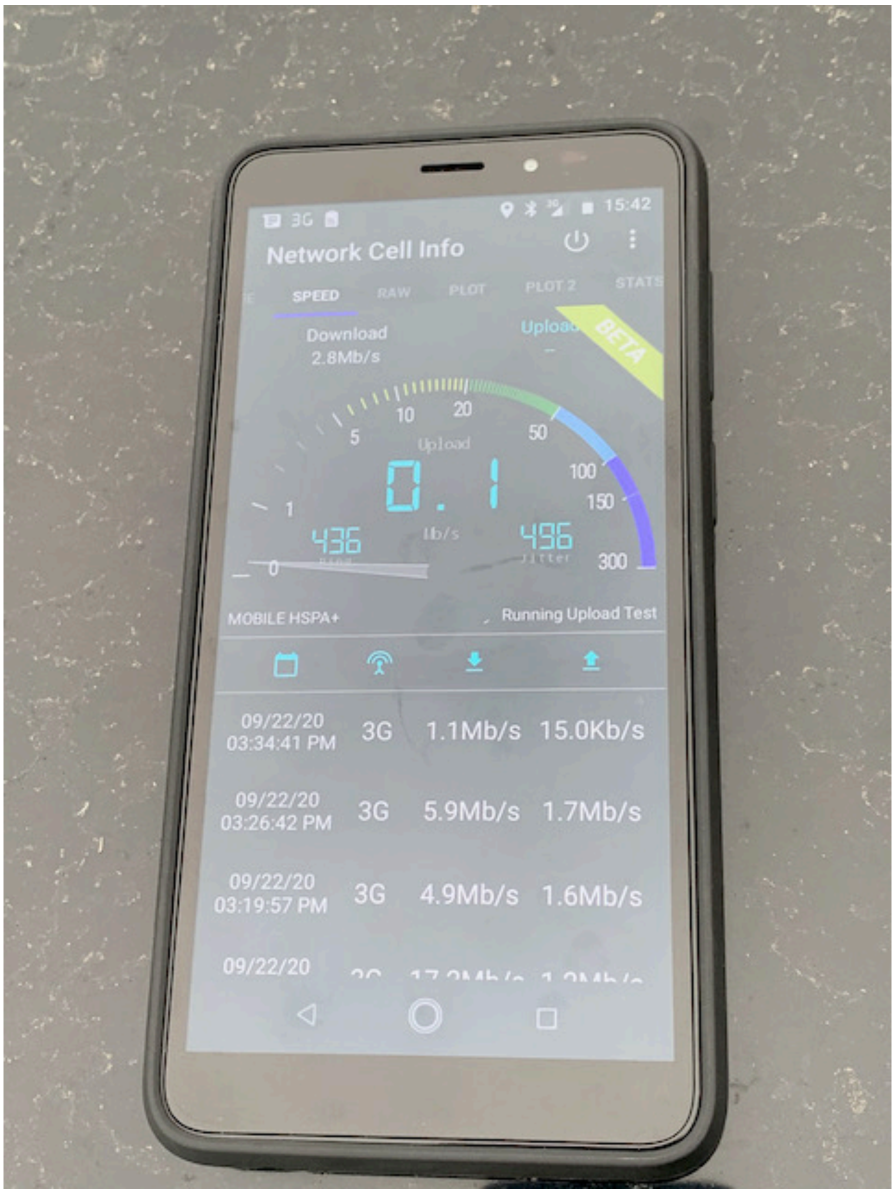
In order to test the operation of the SRUP protocol in conditions of extremely poor network signal, such as may be considered as the extreme of conditions in which a protocol such as SRUP may wish to be used; network conditions simulating these worst-case conditions were also considered.

A simple real-world measurement was taken within an isolated area of the New Forest National Park, to augment this dataset and provide a representative data point for deployment of a 3G device in a rural location. The New Forest was selected as an area local to Southampton, with poor network coverage in order to record real-world conditions for signal strength and network performance in an area of known poor coverage.

A measurement made using a smartphone 'network cell information' measurement app [325] showed a very poor signal strength of -128 decibel-milliwatts (dBm) of Reference Signals Received Power (RSRP), and 115 kilobits per second (kb/s) upload. A photograph illustrating this is shown in Figure 11.4.

In order to assess performance when operating on slow or narrow-bandwidth connections, the example of legacy Second-Generation (2G) networks, such as GSM (Global System for Mobile Communications) [326], were included too. However, given the lack of published data for the simulation of the network conditions of GSM, representative propagation delay data was taken for 3G networks from Khatouni, Trevisan, and Giordano: with the data transfer rates capped at the 'best cases' for both the Enhanced Data Rates for GSM Evolution (EDGE) [327] and General Packet Radio Service (GPRS) [328] standards.

[2]https://tcconfig.readthedocs.io/



09/22/20 03:34:41 PM	3G	1.1Mb/s	15.0Kb/s
09/22/20 03:26:42 PM	3G	5.9Mb/s	1.7Mb/s
09/22/20 03:19:57 PM	3G	4.9Mb/s	1.6Mb/s
09/22/20	[illegible]	[illegible]	[illegible]

FIGURE 11.4: A screenshot from the mobile application used to assess cellular network performance in a rural area of Southern England

Finally, in order to establish a plausible ‘worst case’, packet loss data taken from Ghaderi and Boutaba [329] were applied to the network conditions simulation, in addition to the network propagation time delays.

11.3.3 Experimental conditions

The configuration for the experimental runs were as shown in Table 11.2, and the full details of the network condition parameters used for each of these can be found in Appendix E.

Each of these conditions were run as an experiment, with all five devices running the SRUP protocol.

Experiment	Condition
1	LAN Ethernet (no network conditioning)
2	‘Good’ strength 4G
3	‘Medium’ Strength 4G
4	‘Good’ 3G
5	‘Poor’ 3G
6	2G EDGE ‘best case’
7	Observed 3G ‘poor’ signal (including a capped data rate)
8	2G GPRS ‘best case’
9	3G ‘poor’ signal + 5% loss
10	2G GPRS + 10% loss

TABLE 11.2: Network conditioning settings for each of the ten cases used for the experimental assessment of the SRUP protocol.

For the comparison experiment (using the MQTT protocol), a representative sample of five of the network conditions were examined: the ‘best case’, reasonable ‘worst case’, and three cases in between (cases 1, 2, 3, 5, & 9). These five cases, where both SRUP and MQTT were run, will be referred to as the ‘combined experiments’.

11.4 Experimental hypothesis and measurements

When using the SRUP protocol, an increase in the time taken for message processing was expected, along with increase in the power consumption of the device (both due to the additional processing requirements of running the message signing algorithms). The total size of the data sent was also expected to be increased (due to the additional fields used by SRUP to ensure message security).

When conducting the experimental runs, the following measurements were taken:

1. Time
   The actual performance measurement has been assessed by analysis of the log files produced by the devices, and the server for any given experimental run. Full details of this analysis can be seen in Section 11.5.
2. Power
   Assessment was made of the average power consumption of one of the devices when running both the MQTT and SRUP conditions. Measurement was made using a logging USB power meter.
3. Message Size
   The network traffic was captured using Wireshark [330] and examined to identify the size of the raw MQTT and the SRUP implementation’s MQTT messages.

11.5 Analysis

All of the log file analysis was performed using Python and Jupyter notebooks [331]. The *pandas* library [332] was used for 'data wrangling' and collation.

This analysis involved:

1. Mapping device ID to logical device number
2. Loading the log files from the C2 server for each experiment
3. Stripping the unused columns out of the resulting dataframe
4. Loading all of the log files from each device, for each experiment, and combining them into a single Python object (a list of dictionaries of dataframes).
5. Generating a new dataframe for each row in the C2 log dataframe, recording the device number, the type of operation (on or off), and the timestamp at which the command was sent
6. For each row in the dataframe generated in step 5, extracting the timestamp at which this command was received by the device
7. Calculating the time delay between sending and receiving, in milliseconds

Each experiment generated a graph, which was used to check the data ingestion process (exemplars shown in Figures 11.5 and 11.6). As expected, experiments where a delay distribution had been applied had a significantly greater standard deviation. A similar analysis process was also conducted for the experimental runs using MQTT.

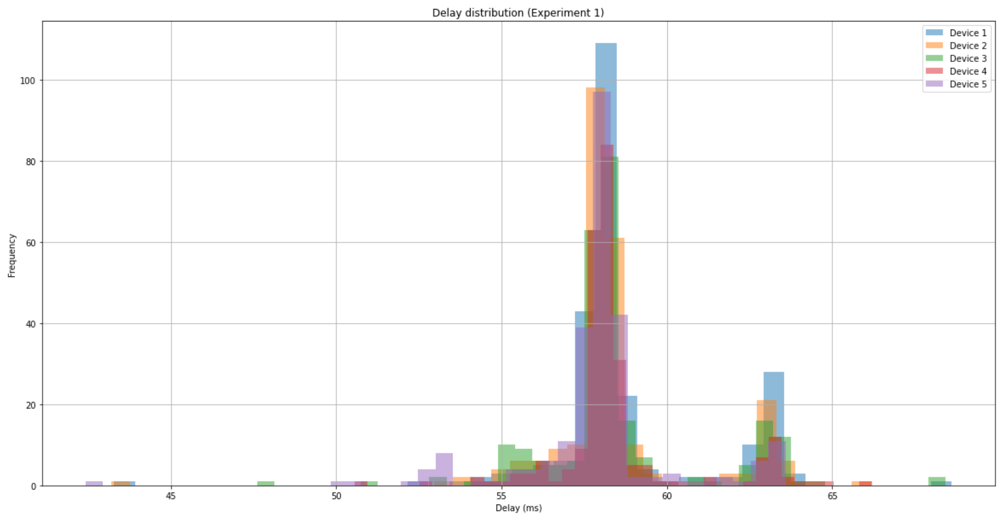


FIGURE 11.5: A graph showing the delay distribution associated with SRUP message propagation and processing time, for experiment 1 (no network conditioning).

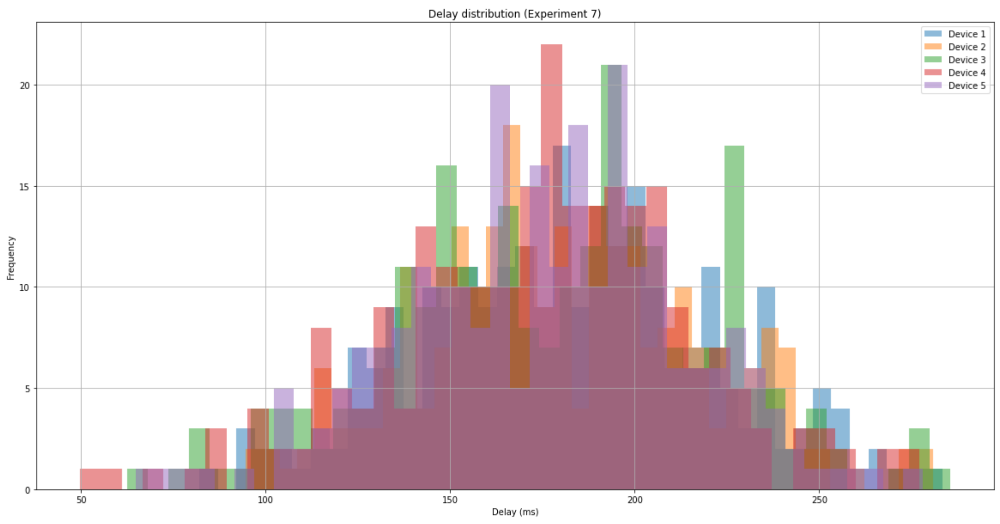


FIGURE 11.6: A graph showing the delay distribution associated with SRUP message propagation and processing time, for experiment 7 (Observed 3G poor signal).

Once complete, the next step of the analysis process was to calculate the mean delay for each device, for each experiment. The means for each device were then averaged in order to calculate the combined mean for each experiment, and the total processing overhead for each experiment could then be calculated. Full details of the analyses can be seen in the Jupyter notebooks in [333].

11.6 Results

11.6.1 SRUP vs. MQTT performance comparison

A graph showing the mean delay for each of the SRUP experiments is shown in Figure 11.7, and a graph showing the combined means for each device for a given experiment (for both the SRUP and MQTT cases) is shown in Figure 11.8. The difference between the mean delays for each protocol is shown in Figure 11.9.

The total combined mean processing overhead for SRUP, when compared with MQTT, across all network conditions, was shown to be an additional 51.60 ms. This compares to the worst-case of 56.13 ms when excluding the effects of the network delay (experiment 1). Although on an Ethernet LAN this represents a significant additional delay (58.44 ms vs. 2.308 ms) compared with a wholly insecure system, when compared with a more representative scenario for deployed IoT (experiment 3, medium strength 4G): the overhead represents only 53.55 % of the MQTT delay (147.7 ms vs. 96.17 ms = 51.53 ms)

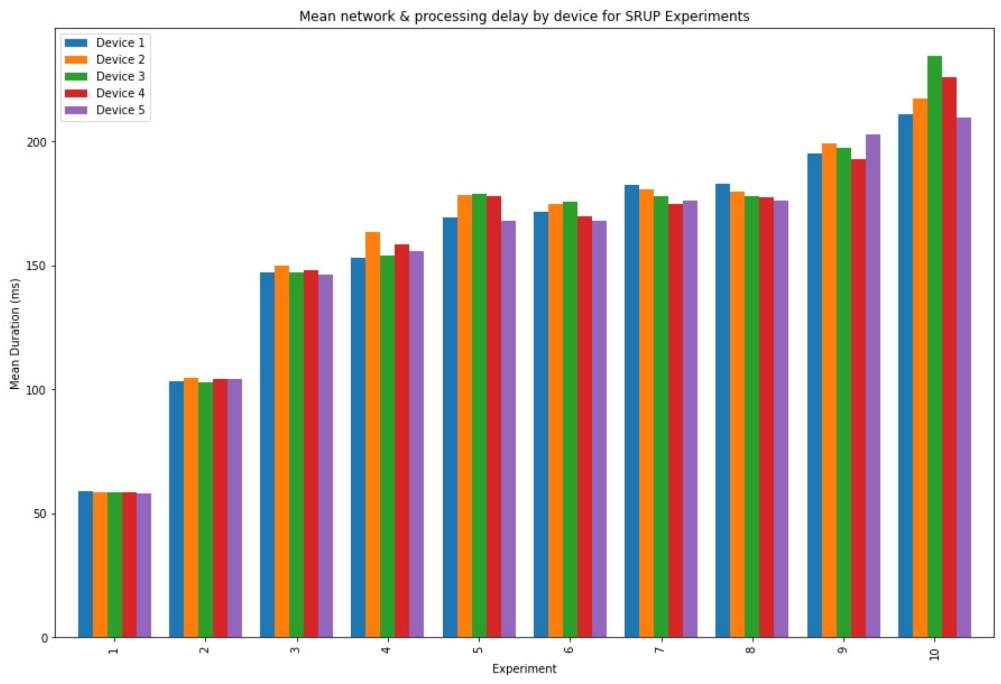


FIGURE 11.7: A graph showing the mean SRUP message network and processing delay by device, for each of the experiments.

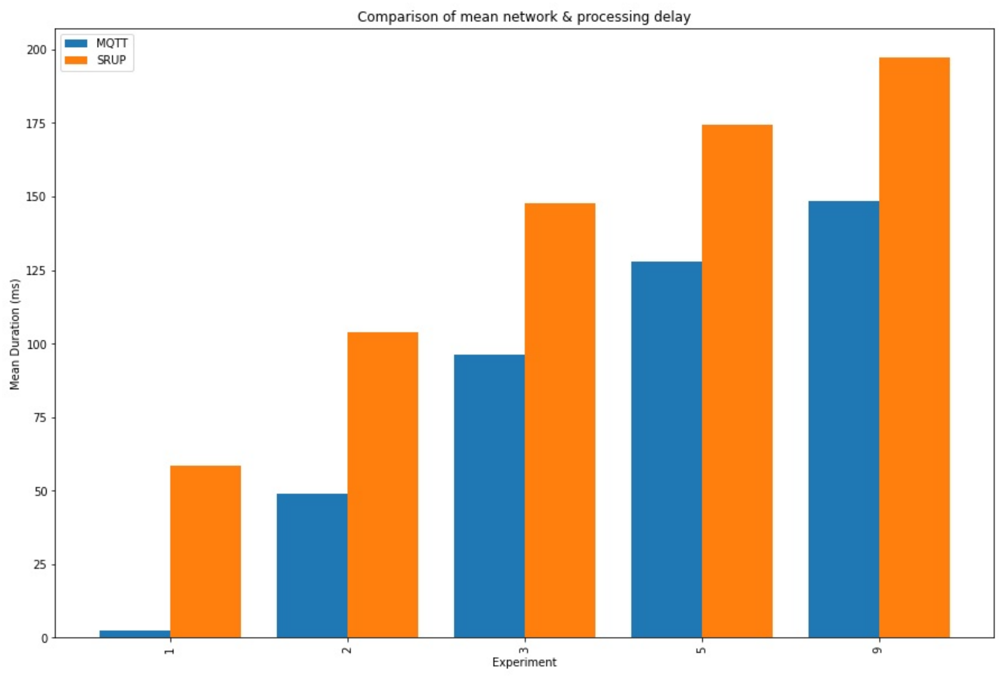


FIGURE 11.8: A graph showing the total mean network and processing delay for both MQTT and SRUP messages, in the five combined experiments (experiments 1, 2, 3, 5, & 9).

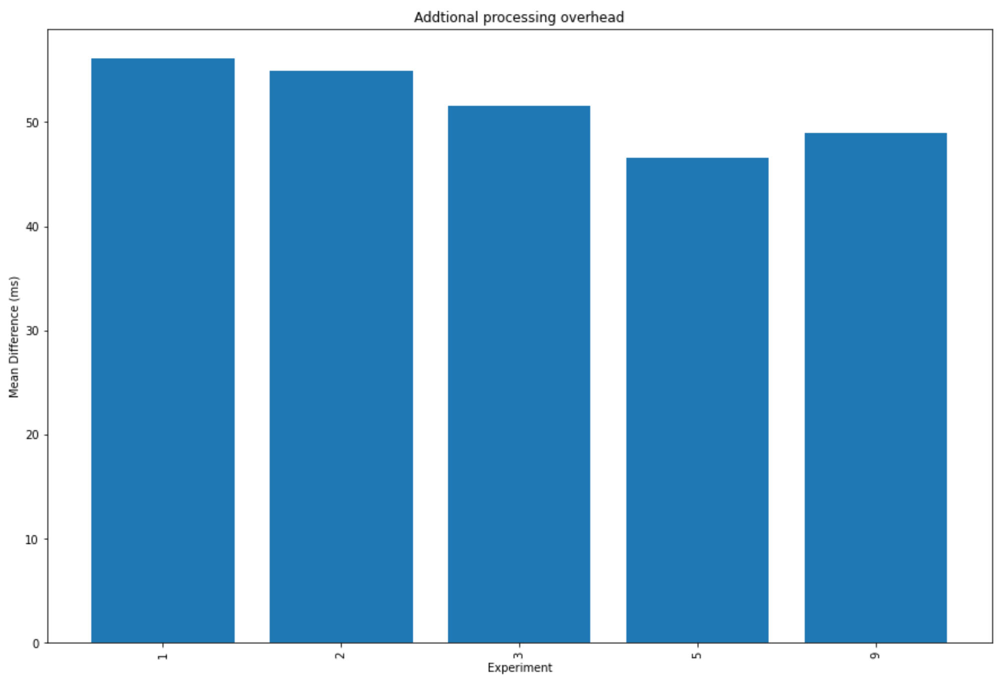


FIGURE 11.9: A graph showing the difference between the total mean network and processing delay for the five combined experiments.

Even in the worst-case, the processing overhead means that only where a message frequency exceeds 17.82 Hz, will the additional processing time be greater than the natural message period (Equation 11.1).

$$\frac{1}{56.13\,\text{ms}} = 17.82\,\text{Hz} \tag{11.1}$$

Since a typical real-world IoT device may be expected to have a mean time between messages of minutes, the additional processing overhead in the order of tens of milliseconds is a very small additional price to pay for the very significant security benefits that the SRUP protocol offers.

However this result does show that the protocol in its current form may not be well suited to highly time-critical applications, when running on lower-specification hardware.

A boxplot graph depicting the extent of the distribution of the data across all of the combined experiments (and showing the minima, maxima, median and 1st & 3rd quartiles) is shown in Figure 11.10.

The analysis also shows that the SRUP protocol is robust to even extremely poor network conditions. Even in the worst case (case 10) all messages were correctly received within 4026 ms ($\sigma = 182.7$ ms, $\bar{x} = 219.6$ ms) (as shown in Figure 11.11) due to the robust nature of the underpinning MQTT protocol.

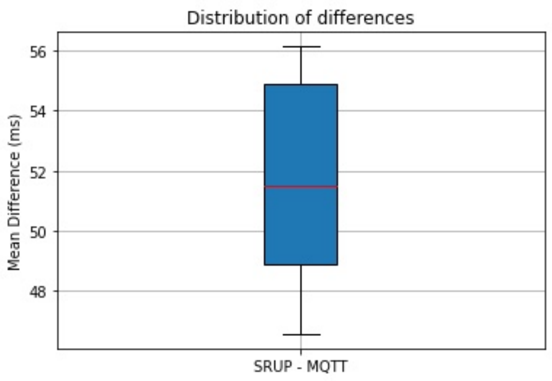


FIGURE 11.10: A box-plot chart showing the distribution of differences in the network and processing delay between the MQTT and SRUP experiments, considered over all of the combined experiments.

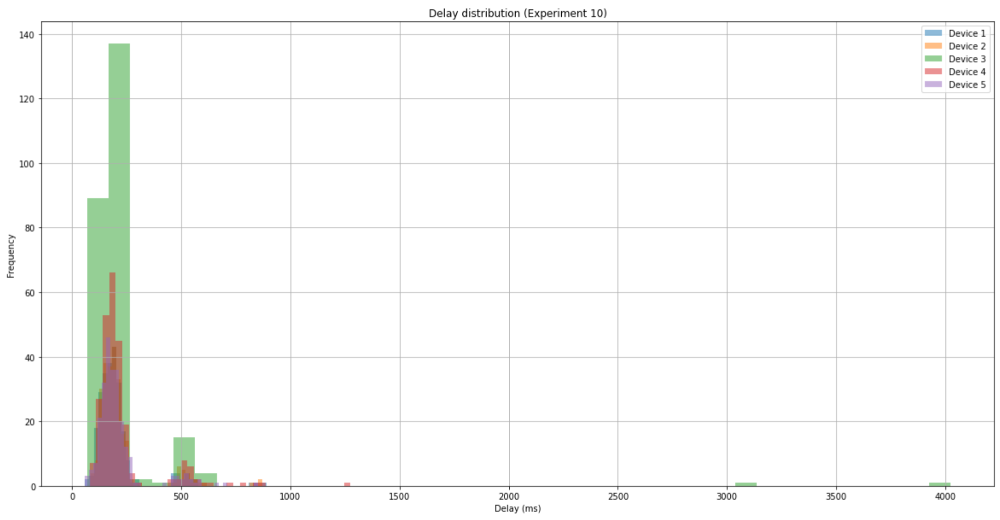


FIGURE 11.11: A graph showing the delay distribution associated with message propagation and processing time, for experiment 10 (2G GPRS + 10% packet loss).

11.6.2 Raspberry Pi 3B+ vs. Raspberry Pi 4

The additional processing power of the *Pi 4* was shown to have a benefit in reducing the overhead incurred by the use of SRUP. The *Pi 4* was, on average, 8.681 ms faster than the *Pi 3B+* when using the SRUP protocol. In comparison, the MQTT protocol was just 0.2738 ms faster in the same context. Thus, the use of a *Pi 4* reduces the total overhead processing delay to 42.92 ms for SRUP.

11.6.3 SRUP vs. MQTT power consumption

A graph showing the instantaneous power consumption for a Raspberry Pi 3B+ device, running the combined experiment (network conditions 1) is shown in Figure 11.12.

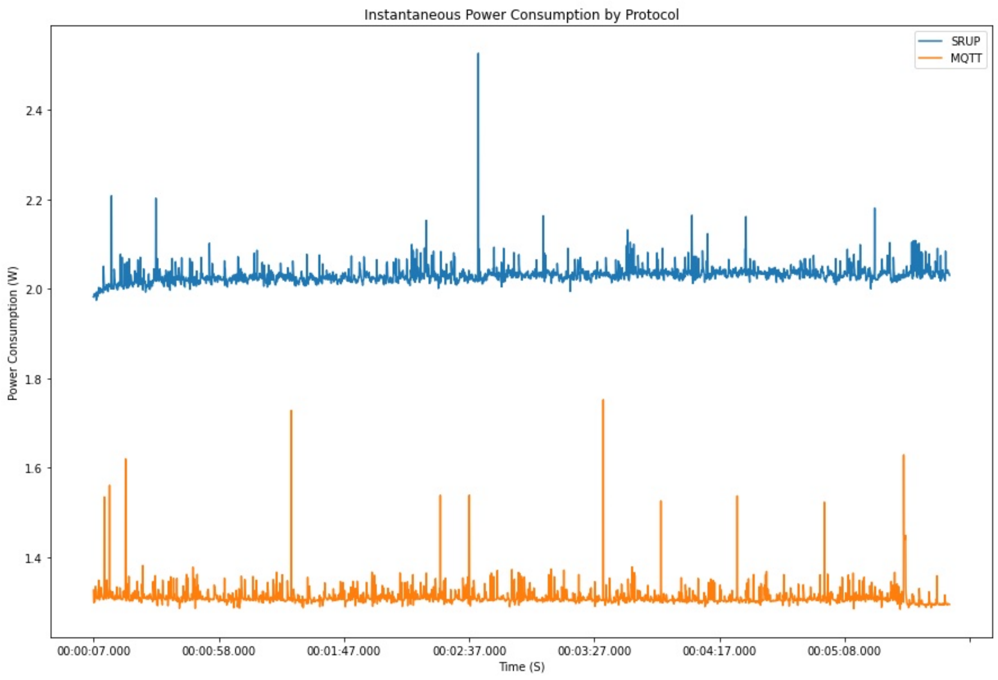


FIGURE 11.12: A graph showing the instantaneous power consumption of the experimental device, by time, for a combined experiment (using network conditions 1).

The mean power consumption for the MQTT run is 1.312 W ($\sigma = 0.04007$ W), which compares to a mean power consumption of 2.039 W ($\sigma = 0.04703$ W) for SRUP. The power consumption of the device when running SRUP can thus be shown to be an additional 727.6 mW when compared with MQTT.

This represents an increase in power-consumption of 55.47 %. (Equation 11.2).

$$\frac{727.6\,\mathrm{mW}}{1.312\,\mathrm{W}} \times 100 = 55.47\,\% \tag{11.2}$$

If the device was powered over USB (at 5 V) from a 10 000 mA h battery, the energy of the battery may be expressed as $(10{,}000 \times 5)\,/\,1000 = 50\,\mathrm{W\,h}$.

$$\frac{50\,\mathrm{W\,h}}{1.312\,\mathrm{W}} = 38.14\,\mathrm{h} \tag{11.3}$$

$$\frac{50\,\mathrm{W\,h}}{2.039\,\mathrm{W}} = 24.52\,\mathrm{h} \tag{11.4}$$

$$38.14\,\mathrm{h} - 24.52\,\mathrm{h} = 13.62\,\mathrm{h} \tag{11.5}$$

We can calculate (Equations 11.3, 11.4, & 11.5) that the battery would be expected to last for 38.14 hours of continual MQTT activity, versus 24.52 hours of SRUP operation: a difference of 13.62 hours of continuous operation. A real-world IoT application, however, would be very unlikely to be operating in a state of continuous message exchange, and in may typically exchange at most a few messages per minute, therefore the additional power required to process the messages represents only a very small proportion of the overall power consumption in use.

11.6.4 SRUP vs. MQTT message size

Analysis of the Wireshark traffic capture shows the differences between the message lengths. For the MQTT setup, a single text character was sent as the message payload using either an ASCII (American Standard Code for Information Interchange) `‘1’` or a `‘0’` (ASCII 31 or 30), corresponding to the *on* or *off* operation. This, combined with the MQTT topic used to identify the destination device (e.g. `test/d1`), results in a message size of 80 bytes.

In comparison, for the SRUP experiment the same signal was sent using a message comprised of:

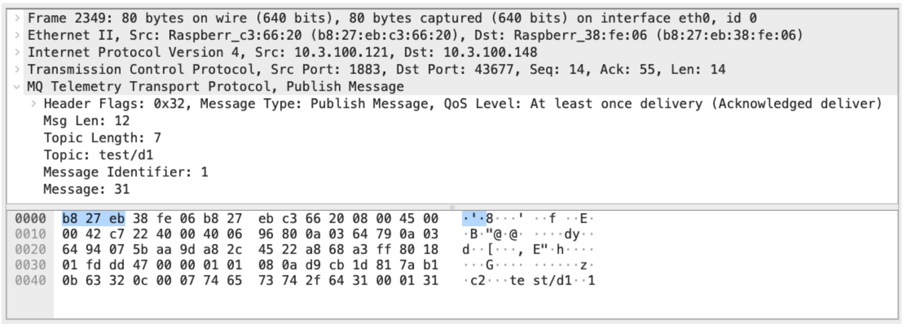
- One byte (`0x00` or `0xFF`) to signify the operation to perform (*on* or *off*)
- A two-byte SRUP message header
- An eight-byte sequence ID
- An eight-byte sender ID
- A variable-length token
- The RSA signature

The message is sent to an MQTT topic corresponding to the device ID prefixed by the word `SRUP`. This results in a SRUP message size of 359 bytes, and an overall TLS packet length of 430 bytes. This represents an approximately 540% increase in data for the SRUP application when compared with MQTT.

An example dataframe corresponding to each of these two message types can be seen in Figure 11.13.

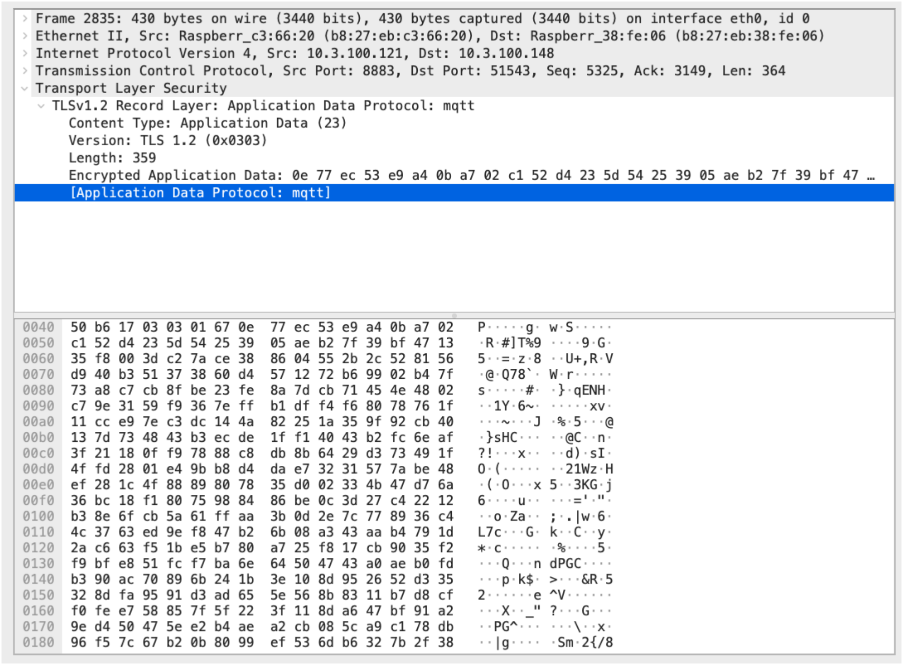
11.7 Evaluation of results

The experimental data and subsequent analysis has shown that the overhead associated in processing messages sent using SRUP, in comparison with insecure MQTT messages, is independent of the network conditions. SRUP has also been shown to have an overhead that is tolerable for all messaging applications bar those that are the most time-sensitive or those requiring a higher message frequency than 17.8 messages-per-second.



```
› Frame 2349: 80 bytes on wire (640 bits), 80 bytes captured (640 bits) on interface eth0, id 0
› Ethernet II, Src: Raspberr_c3:66:20 (b8:27:eb:c3:66:20), Dst: Raspberr_38:fe:06 (b8:27:eb:38:fe:06)
› Internet Protocol Version 4, Src: 10.3.100.121, Dst: 10.3.100.148
› Transmission Control Protocol, Src Port: 1883, Dst Port: 43677, Seq: 14, Ack: 55, Len: 14
⌄ MQ Telemetry Transport Protocol, Publish Message
  › Header Flags: 0x32, Message Type: Publish Message, QoS Level: At least once delivery (Acknowledged deliver)
    Msg Len: 12
    Topic Length: 7
    Topic: test/d1
    Message Identifier: 1
    Message: 31

0000  b8 27 eb 38 fe 06 b8 27  eb c3 66 20 08 00 45 00   ·'·8···' ··f ··E·
0010  00 42 c7 22 40 00 40 06  96 80 0a 03 64 79 0a 03   ·B·"@·@· ····dy··
0020  64 94 07 5b aa 9d a8 2c  45 22 a8 68 a3 ff 80 18   d··[···, E"·h····
0030  01 fd dd 47 00 00 01 01  08 0a d9 cb 1d 81 7a b1   ···G···· ······z·
0040  0b 63 32 0c 00 07 74 65  73 74 2f 64 31 00 01 31   ·c2···te st/d1··1
```

(A) The raw data from an MQTT message. Note that the data is unencrypted and visible as cleartext.



```
› Frame 2835: 430 bytes on wire (3440 bits), 430 bytes captured (3440 bits) on interface eth0, id 0
› Ethernet II, Src: Raspberr_c3:66:20 (b8:27:eb:c3:66:20), Dst: Raspberr_38:fe:06 (b8:27:eb:38:fe:06)
› Internet Protocol Version 4, Src: 10.3.100.121, Dst: 10.3.100.148
› Transmission Control Protocol, Src Port: 8883, Dst Port: 51543, Seq: 5325, Ack: 3149, Len: 364
⌄ Transport Layer Security
  ⌄ TLSv1.2 Record Layer: Application Data Protocol: mqtt
      Content Type: Application Data (23)
      Version: TLS 1.2 (0x0303)
      Length: 359
      Encrypted Application Data: 0e 77 ec 53 e9 a4 0b a7 02 c1 52 d4 23 5d 54 25 39 05 ae b2 7f 39 bf 47 …
      [Application Data Protocol: mqtt]

0040  50 b6 17 03 03 01 67 0e  77 ec 53 e9 a4 0b a7 02   P·····g· w·S·····
0050  c1 52 d4 23 5d 54 25 39  05 ae b2 7f 39 bf 47 13   ·R·#]T%9 ····9·G·
0060  35 f8 00 3d c2 7a ce 38  86 04 55 2b 2c 52 81 56   5··=·z·8 ··U+,R·V
0070  d9 40 b3 51 37 38 60 d4  57 12 72 b6 99 02 b4 7f   ·@·Q78`· W·r·····
0080  73 a8 c7 cb 8f be 23 fe  8a 7d cb 71 45 4e 48 02   s·····#· ·}·qENH·
0090  c7 9e 31 59 f9 36 7e ff  b1 df f4 f6 80 78 76 1f   ··1Y·6~· ·····xv·
00a0  11 cc e9 7e c3 dc 14 4a  82 25 1a 35 9f 92 cb 40   ···~···J ·%·5···@
00b0  13 7d 73 48 43 b3 ec de  1f f1 40 43 b2 fc 6e af   ·}sHC··· ··@C··n·
00c0  3f 21 18 0f f9 78 88 c8  db 8b 64 29 d3 73 49 1f   ?!···x·· ··d)·sI·
00d0  4f fd 28 01 e4 9b b8 d4  da e7 32 31 57 7a be 48   O·(····· ··21Wz·H
00e0  ef 28 1c 4f 88 89 80 78  35 d0 02 33 4b 47 d7 6a   ·(·O···x 5··3KG·j
00f0  36 bc 18 f1 80 75 98 84  86 be 0c 3d 27 c4 22 12   6····u·· ···='·"·
0100  b3 8e 6f cb 5a 61 ff aa  3b 0d 2e 7c 77 89 36 c4   ··o·Za·· ;·.|w·6·
0110  4c 37 63 ed 9e f8 47 b2  6b 08 a3 43 aa b4 79 1d   L7c···G· k··C··y·
0120  2a c6 63 f5 1b e5 b7 80  a7 25 f8 17 cb 90 35 f2   *·c····· ·%····5·
0130  f9 bf e8 51 fc f7 ba 6e  64 50 47 43 a0 ae b0 fd   ···Q···n dPGC····
0140  b3 90 ac 70 89 6b 24 1b  3e 10 8d 95 26 52 d3 35   ···p·k$· >···&R·5
0150  32 8d fa 95 91 d3 ad 65  5e 56 8b 83 11 b7 d8 cf   2······e ^V······
0160  f0 fe e7 58 85 7f 5f 22  3f 11 8d a6 47 bf 91 a2   ···X··_" ?···G···
0170  9e d4 50 47 5e e2 b4 ae  a2 cb 08 5c a9 c1 78 db   ··PG^··· ···\··x·
0180  96 f5 7c 67 b2 0b 80 99  ef 53 6d b6 32 7b 2f 38   ··|g···· ·Sm·2{/8
```

(B) The raw data from a SRUP message, encrypted using Transport Layer Security.

FIGURE 11.13: The raw network data for two messages (one an MQTT and the other a SRUP message) carrying the same data, captured in Wireshark

The benefits of using SRUP are highly significant for any real-world application of IoT technologies. Without message encryption, the data contained within the messages may be freely obtained by anyone with access to the transport network or anyone who is able to *sniff* traffic leaving or entering the devices or servers. Authentication ensures that messages are protected from both deliberate tampering or accidental corruption in transit, and ensures that only validated and approved senders can issue commands to devices. The SRUP protocol offers protection against replay attacks by removing the possibility of an attacker capturing a valid message. Such protections greatly enhance the security of IoT systems, rendering them much more suitable for use in applications such as building management or monitoring. The processing delay that is required to provide these benefits is small, and acceptable for devices running on hardware with higher-performance CPUs.

Although there is a cost in terms of the power consumption of devices using SRUP, this is only an issue for battery-powered devices which have a high message rate.

Although the data packet size of SRUP messages is significantly larger when compared to MQTT, these experiments have shown that SRUP traffic is robust to even extremely poor network conditions and that messages are still successfully delivered in a timely manner. This increased packet size does, however, mean that the protocol is not well-suited for use with extremely constrained bandwidth communications bearers, such as LoRa [334] or SigFox [335].

11.8 Summary

This Chapter has described the experimental assessment of the SRUP protocol, and the subsequent analysis of the data produced. It has shown that the SRUP protocol (when compared with a simple, insecure, MQTT implementation) has a total message processing delay of between 51.60 ms and 42.92 ms when running on Raspberry Pi hardware, and an increased power consumption of 727.6 mW whilst processing messages.

The next and final Chapter concludes the work, summarizing the contributions to knowledge made by this research, and answering the Research Questions posed in Chapter 1. It also makes some recommendations for follow-on work to enable the further exploitation of this research.

Chapter 12

Conclusions

This final Chapter describes the conclusions of this work. It seeks to identify the contribution made by this research, and to discuss how it has answered the research questions posed at the beginning of this thesis. It also discusses some recommendations for future work and describes the potential future impact of the work.

A short summary video that demonstrates the implementation of many of the aspects of this research can be seen at: https://youtu.be/F0_qlqh0Oiw [319].

12.1 Answering the Research Questions

Section 1.1.1 introduced the seven research questions, which this work has sought to address. In this Section, these will be examined in turn and answered.

12.1.1 Research Question 1

How can a secure protocol for Command and Control messaging for the Internet of Things be developed from extant, tried and tested, commodity software and network communications components?

The Secure Remote Update Protocol provides an efficient design and implementation for secure C2 messaging for IoT applications. All of the underpinning elements for SRUP are off-the-shelf components. It has been built on top of the very widely used MQTT protocol (which itself runs over TCP). It exploits TLS (and within that, AES) for message encryption, and (in its reference implementation) utilizes RSA and SHA-256 for message signing. This adoption of well known components lends some credibility to the protocol, and makes it easier for users to understand. The widespread use of these components also helps to ensure that they are robust and error-free — which in turn makes it easier to ensure the successful operation of SRUP.

12.1.2 Research Question 2

> *How can such a protocol be used to enable automated secure key distribution and identity management?*

SRUP adopts the paradigm of dynamic identity. This, combined with the *self-service* device registration and key exchange model adopted, ensures that no human intervention is required to support the process of a device registering with a SRUP universe once the device in question has received the registration URL for that universe. The use of MQTT broker access control measures such as an ACL and the careful use of MQTT topics ensure that messages can only be sent by valid devices, and only received by their intended recipients. The semi-automated process for C2 server registration (requiring access to the SRUP backend) ensures that only correctly approved C2 servers may be added to a universe, but that once they have been, the SRUP backend system automatically provides the necessary identity management to support devices joining that C2 server.

12.1.3 Research Question 3

> *How can such a protocol (and associated architectures) be used to provide assurance around the identity of a physical device?*

The use of dynamic identity means that external mechanisms are required to ensure that the logical identity of a given physical device can be proven. The utilization of the observed join process enables the provision of such guarantees, either supported by a trusted human user (manually comparing proof-of-identity codes, using one of the visual techniques such as pictograms), or supported by an automated observer device reading the code via computer vision techniques (such as QR codes) or Near-Field Communication. Adopting this approach means that for systems utilizing SRUP there is no requirement for fixed identities. This both reduces the complexity of building compatible devices and permits easier and more thorough revocation of identity to ensure security when devices change ownership.

12.1.4 Research Question 4

Can the protocol be made robust to attempted attacks against it, and its supporting infrastructure?

SRUP has been designed from the outset of this research work to be security-focused. The message-signing and encryption ensure confidentiality, authenticity and integrity of the data and wider systems built using SRUP. The protocol is resistant to replay attacks, and the support for authenticated software update means that systems utilizing this approach are resistant to malicious remote update attacks.

The designed in crypto-agility means that although any specific implementation may be subject to future attacks against the cryptosystem in use, the protocol itself may simply be reimplemented to use any other asymmetric cryptosystem that supports message signing and encryption.

12.1.5 Research Question 5

How can the protocol be made sufficiently easy to use, that it becomes simpler for a prospective user to adopt the secure protocol, than to implement an insecure system?

Implementing secure systems using the SRUP protocol can be accomplished by using the abstracted code libraries such as `pySRUP`. By adopting this approach a user may develop complex systems whilst writing the minimum of code to handle the implementation of the protocol itself. Although at present the library code must be built locally (and isn't available for direct installation via public library repositories and package management systems), the use of automated build scripts makes this process very straightforward. Together with the containerized backend systems, and the key generation tool to bootstrap a new setup, the barrier of entry to using the SRUP protocol is extremely low.

12.1.6 Research Question 6

Can such a protocol be extended to provide a mechanism to securely share controlled subsets of data between Command and Control systems, whilst enabling system owners to retain overall control?

The syndication process developed during this work permits the cooperation between different C2 systems — even if they operate within different SRUP universes (and regardless of whether or not they use the same underlying cryptosystems). This will enable devices utilizing SRUP to share partial information (and the option for moderated control) with other C2 systems which are not fully trusted by the providing system. The ability to utilize this approach in the context of the provision of guest access to IoT devices in commercial and domestic settings is potentially a very important element in the solution of the problem of how to permit guest users within increasingly automated systems.

12.1.7 Research Question 7

What is the performance overhead of such a protocol when compared to insecure methods, and how well does it cope with poor network conditions?

The experimental evaluation of the protocol has shown that SRUP operates successfully, even in extremely poor network conditions (far beyond those which would typically be seen in the vast majority of real-world deployments). The experimentation showed that although there is a performance overhead associated with using the protocol, this is a relatively small factor for most realistic use-cases. This overhead is processor dependent and so the utilization of faster CPU will reduce this cost. There is an associated increase in power consumption when using the secured protocol (with its greater CPU utilization), however it is not expected that many deployed (and battery powered) devices would use continuous message exchange in real-world applications. Although message sizes (and correspondingly the data rates) are necessarily larger when using the encrypted and secured protocol, the overhead in absolute terms is still very small.

12.2 Contributions to knowledge

This research has introduced a number of original elements. These are highlighted in this Section.

12.2.1 The Secure Remote Update Protocol

The Secure Remote Update Protocol itself represents an entirely new protocol, defining a schema for the payload of MQTT messages and the associated protocol rules in terms of message handling, validation and C2 network operations. The protocol has been implemented and evaluated experimentally, and has been shown to work with only a small overhead processing delay of 51.60 ms.

12.2.2 Dynamic identity and key management

The adoption of dynamic identity for IoT devices reduces the complexity of device production and facilitates identity revocation and regeneration to ensure that device access by legacy users can be prevented. The automated key management process means that devices can register automatically and without recourse to human intervention until they are being added to a particular C2 network which may require this to perform a join operation. This separates the registration and deployment into two discrete phases, and which use different networking protocols to complete.

12.2.3 Device identity validation

The use of dynamic identity and the automated key exchange process is made possible by the use of the observation process to relate the physical object to its logical identity. The use of techniques to aid human comparison of secure identity values such as wordlists and pictograms in this context is innovative, as is the use of machine-based observation processes.

12.2.4 Software library implementation and containerized backend systems

The software implementation in the form of an easy to use library enables non-specialists to implement secure C2 networks. The use of easy to configure containerized backend systems makes it easier to deploy a SRUP system than would otherwise be possible. By turning this into a commodity component, hosting a self-contained universe for C2 networks also becomes something that non-specialists can do, and could be adopted by individuals or organizations with strong privacy concerns.

12.2.5 Syndication

The syndication concept enables sharing of information across organizational or system boundaries, and its applicability as a solution to the problem of guest access is an important innovation.

12.3 Recommendations for future work

Although the current research project is now complete, a number of additional areas have been identified where follow-on work can be recommended.

12.3.1 SRUP for microcontrollers

Although all of the work conducted during this research has considered IoT devices built using single-board computers running a full version of the Linux Operating System, the protocol itself does not require such a system, and could be implemented on much simpler microcontroller-based devices. Such devices would not-only likely have better power consumption and a lower cost, but due to them not running a full OS, may be more resilient to some types of attack (exploiting wider vulnerabilities within the Linux OS). Therefore further research is recommended to evaluate the performance characteristics, security and power consumption demands of running SRUP on battery powered devices that utilize specialized low power hardware and embedded microcontrollers, such as the ESP32[1], or RP2040[2]. Although traditionally microcontroller based devices have not been well-suited to algorithms utilizing asymmetric cryptography (due to the processing demands of such techniques), modern ARM-based microcontrollers such as the RP2040 (which utilizes an ARM Cortex-M0 processor) feature multi-core 32-bit CPUs; and others (such as the ESP32), include dedicated cryptographic co-processor hardware for greater performance.

12.3.2 Alternative transports for the SRUP protocol

Whilst the SRUP protocol was designed around using MQTT as the *application transport layer* for exchanging data between devices, a revised version of the protocol could be created to utilize alternative transport. Such an approach may be especially well suited to devices designed to be deployed for greatly extended periods of time — and which would be designed to spend much of their deployment in low-power *sleep* modes where power consuming peripherals such as RF network interfaces would be turned off for much of the time. It is recommended that future work be conducted to explore approaches that could be applied in order to extend the concept of SRUP to this class of device.

12.3.3 SRUP Syndication for guest access

The concept of using syndication to facilitate guest access for IoT has not been explored experimentally. It is recommended that research to evaluate syndication specifically in this context be conducted, along with related work to examine how SRUP and syndication could be utilized in the context of contemporary domestic and commercial IoT devices and the hub-based control model of automation and *digital personal assistants*.

12.3.4 Real-world evaluation of machine-based observation

This research established proof-of-concept demonstrations of machine-based observation to establish device identity. However it is recommended that experimental work be conducted to evaluate the concept in real-world scenarios. These could include both the

[1] https://esp32.com
[2] https://www.raspberrypi.org/products/raspberry-pi-pico/

use of SRUP for the management of automated entities joining a C2 network, and a scenario for deploying devices at scale, where an operator could initiate the joining of the device to the C2 system, and then confirm its identity using a hand-held reader or smartphone.

12.3.5 Human comparison of security identifiers

Although a number of approaches to enable easy human comparison of large cryptographic identity values were described within this research, it is recommended that further work be conducted to evaluate human-performance in the context of correctly discriminating between accidental and deliberate mismatches. This research should conduct assessments of both speed and error-rate of comparisons using a range of different visualization and presentation techniques, in a range of human subjects.

12.3.6 Publication of binary version of SRUP library

Although the software implementations of SRUP (such as the `pySRUP` library) are publicly accessible, the software must be built from the supplied source code, as no extant binary distributions are available. In order to encourage more widespread use of the protocol, it is recommended that additional work is conducted to make such binary distributions available, and in particular to publish a version of `pySRUP` to the Python Package Index repository[3] so that it can be automatically installed by potential users, using standard Python package management tools (such as `pip`). Additional work would also be required to produce suitable supporting documentation for the library, backend, and key generation tools.

12.3.7 Application of SRUP in related domains

SRUP was designed for use in the context of C2 messaging for the Internet of Things, but the protocol is potentially also well suited for use in wider domains where C2 messages are required to be sent from a server to a device. These could include autonomous uncrewed vehicles, semi-autonomous *platooned* vehicle convoys, or highway smart signalling applications. Research should be conducted to understand the particular requirements of these applications (such as timeliness, latency, and message volume), and to then assess the viability of utilizing SRUP for these types of use-case.

[3]https://pypi.org

12.4 Potential future impact of the research

Although development builds of the SRUP software have been publicly available since the beginning of this research, the final version has only recently been completed —– and as such its use beyond this research is not widespread at this time.

If more widely adopted however, the research described herein could provide a very significant benefit to the security of IoT devices, and other types of CPS for which remote C2 operations are required, as well as making the development of such devices (and their secure operation) accessible to a much larger group of individuals than is currently the case.

12.5 Summary

This research has proposed, implemented, and experimentally validated a number of techniques to provide for the cybersecurity of Command and Control (C2) messaging for the IoT. C2 systems form a key part of civilian, industrial and military applications. The focus of this research has been on ensuring their security, which is essential for their safe and efficient operation.

The Secure Remote Update Protocol has been proposed and described in detail, as a solution to the requirements for secure C2 messaging for IoT devices.

Seven research questions have been answered, covering the development of a protocol, key distribution and management, device identity, the robustness and security of the protocol, ease of use, sharing of data, and assessing the overhead associated with using the protocol.

An Open Source Software implementation of the protocol has been developed, published on GitHub [292], and described; and has been experimentally validated and shown to have an acceptable message processing overhead when compared to insecure communications. The implementation has been focused around ease-of-use for IoT application developers, to provide a tool that facilitates secure communications, and which requires less work from an application developer than an equivalent insecure implementation of IoT communications.

Appendix A

The operation of MQTT in detail

This Appendix describes the operation of the MQTT protocol in more detail. For further detail, please see the MQTT Specification (version 5.0), [211].

A.1 Connection

When a client connects to the broker it sends a MQTT Connect (`CONNECT`) message to the broker. The message headers indicate whether or not the connecting client wishes to specify a Will, the keep alive time for this client, as well as whether or not the client wishes to specify a username and password.

The payload of the `CONNECT` message contains details of the identifier for the client (which the standard requires is unique amongst all other connected clients using that broker at that time), as well as the Will topic and message, and the username and password to use to establish the connection (noting that MQTT is itself unencrypted so these are passed in plaintext unless a TLS connection is used).

Upon successful connection — the broker will send a Connection Acknowledge (`CONNACK`) message back to the client.

A.2 Keep alive

Upon initial connection to the broker, a client must specify a “keep alive” time. This is the maximum permissible interval (in seconds) that may elapse between that client sending one control message (`PUBLISH`, `PUBACK`, etc.) and the next. In the absence of having data to send — the client must send a Ping Request (`PINGREQ`) control message within the time interval.

The standard requires that the broker should consider the client to be disconnected if it has not received any control messages from that client within 1.5 times the keep alive period.

It is possible to specify a zero-value for keep alive to disable this feature for a given client, and the maximum value (given that a 16-bit word is used for the value) is 18 hours, 12 minutes, and 15 seconds.

A.3 Ping request & response

Upon receiving a `PINGREQ` message from a client the broker will respond by sending a ping response message (`PINGRESP`). This enables the client to determine that the broker is still alive: and that the network connection is still active.

A.4 Last will & testament

MQTT also supports a concept informally known as *Last will & testament*. This is well suited to the sorts of unreliable network environments where a client may be inadvertently disconnected by a dropped connection.

To use the Last Will & Testament — a client sends the details of a Will message, as optional fields upon connection with the broker. The broker stores this message until such a time that it receives a `DISCONNECT` message from the client (in which case it discards the Will message) or if the client fails to respond within the Keep Alive time: in which case the Will message is sent. A client wishing to disconnect cleanly must therefore send a `DISCONNECT` message to disconnect from the broker.

The MQTT protocol makes no attempt to define what clients receiving this message should do (nor the format of the message) — this is therefore application specific.

The concept is designed to enable a device connecting to the broker via an unreliable network channel, to signal to other devices on the network that it has lost connection. For example, if a device (with a device ID of `device123`) connects to the broker using a `CONNECT` message, containing the `lastWillTopic` field set to `/device123/STATUS`, and the `lastWillMessage` field set to a suitable message (e.g. “OFFLINE”). In the event that the connection between `device123` and the broker drops, such that the device will no longer respond to `PINGREQ` messages from the broker (and without `device123` having sent a `DISCONNECT` message): the broker will then consider the device to be disconnected, and publish the Will message on the specified topic. Thus any subscriber to that topic will receive the message — and therefore know that the device suffered unintended disconnection.

Optionally the Will message can also have the `lasteWillRetain` field set to `TRUE`, and thus be set to be *retained* by the broker. In this case, any new subscribers joining the Will topic (after the Will message was published) will also receive a copy (e.g. to signal that the device in question is off-line), until such a time that this is cleared by the device manually sending a zero-byte message to the Will topic upon reconnection.

A.5 Publishing a message

To publish a message the client sends a publish message (`PUBLISH`) consisting of a two-byte header which specifies which Quality of Service it wishes to use, identifies the message as a `PUBLISH` message, and specifies the length of the remainder of the message. The message itself contains the topic that the client is publishing to, the packet identifier (if using QoS 1 or 2) and the payload itself.

A.6 MQTT Quality of Service

MQTT supports three QoS settings.

- QoS0: Deliver at most once
- QoS1: Deliver at least once
- QoS2: Deliver Exactly once

The primary difference between these three modes is the relative overhead sent alongside the message to provide assurance that the message has been correctly delivered.

Given the brokered publish / subscribe model — and the fact that MQTT supports multiple different QoS for different connections (i.e. publishing device A could use QoS1 to talk to the broker, whilst device B subscribes at QoS0) the broker can either be the sender or receiver in the following examples.

A.6.1 QoS0

QoS0 is often referred to as *fire and forget*. There are no delivery guarantees beyond those offered by TCP. Messages are not acknowledged — the receiver does not send any acknowledgement message back to the sender. Consequently QoS0 has the smallest overhead.

QoS0 is ideally suited for lossless (or almost lossless) connections — or situations where messages are sent at a high data rate (and therefore where the next message will be received within a short-interval to replace the missed message).

A.6.2 QoS1

QoS1 guarantees that the message will be delivered at least once. At QoS1, messages received by a receiver are confirmed by the use of a publish acknowledgement message (`PUBACK`). The `PUBACK` is sent by the receiver on receipt of a message. If the sender does not receive a `PUBACK` for a message that it has published, it will resend the message. As such it is possible for messages to be retrieved multiple times (if the `PUBACK` message is not delivered).

QoS1 adds a relatively small overhead: the `PUBACK` message is a 4-bytes MQTT message — consisting of a 2-byte header, and a 2-byte message identifier to relate it to the originally published message. This gives a total size of 60-bytes when including the TCP/IP headers.

QoS1 is perhaps the most useful QoS level as it guarantees all messages are delivered, whilst avoiding undue overhead.

A.6.3 QoS2

At QoS2 delivered messages are guaranteed to be delivered exactly once. This is accomplished using a three-part message exchange. After the initial message has been received the receiver will respond by sending a Publish Received message (`PUBREC`) back to the sender. Once the sender has received the `PUBREC`, it then responds by sending a Publish Release Message (`PUBREL`) back to the receiver. Finally the transaction is concluded by the receiver sending a Publish Complete (`PUBCOMP`) message back to the sender. This clearly adds a more significant overhead to the traffic — as each of the control messages equates to a 4-byte MQTT message (which is a 60-byte total packet).

QoS2 is also the slowest QoS level — and is therefore generally only used for situations where reception of duplicate messages would be actively harmful.

A.7 MQTT subscription

When a client subscribes to a topic, it sends a Subscribe message (`SUBSCRIBE`) in which it must specify the topic (or topics) it wishes to subscribe to — and the QoS it wishes to use for the subscription to the topic. Multiple topic subscriptions can be contained within a single `SUBSCRIBE` message. The broker will reply with a Subscription Acknowledgement (`SUBACK`) message confirming the subscription and QoS.

MQTT topics are hierarchical (although their use is application specific, and as such users may elect to use a flat hierarchy with all topics at the top level). Clients may subscribe to specific topic end-points — or to topic parents at any level in the hierarchy. Subscribing to a parent topic also subscribes to all child topics.

Appendix B

The Secure Remote Update Protocol Specification v3.0

B.1 MQTT topics

All messages sent using the Secure Remote Update Protocol will be sent using MQTT topics corresponding to individual devices. When a device connects to the broker within a SRUP system, it shall be required to subscribe to a topic related to its identity — using the form `SRUP/<DEVICE ID>`.

For example if a device has an identity of `747b22fc2a7541d2`, then it shall be required to subscribe to the topic:

`SRUP/747b22fc2a7541d2`.

In order to signal to a C2 Server that a device wishes to apply to `JOIN` that server's C2 network — then it will send a message to a topic (reserved for use only for the initial `JOIN REQUEST` message) in the form of `SRUP/servers/<SERVER ID>/<DEVICE ID>`.

For example if the device with an identity of `747b22fc2a7541d2` wishes to send a `JOIN REQUEST` to a server with an identity of `b9d077e223834cf6`: then it will use the MQTT topic:

`SRUP/servers/b9d077e223834cf6/747b22fc2a7541d2`.

Upon accepting the `JOIN REQUEST` the server shall be required to subscribe to the device's topic.

All subsequent messages to or from the device shall be sent via the device's topic.

B.2 Update messages

The SRUP Update Messages are provided to initiate and trigger a software update operation. The Server must begin by sending the `INITIATE` message to the Device. On receipt of an `INITIATE` message the Device must attempt to retrieve the update data from the specified URL. The Device must then send a SRUP Response message to the Server to indicate the outcome of the retrieval operation. If the status `SRUP_UPDATE_SUCCESS` is received the Server may then send an `ACTIVATE` message to signal that the Device should apply the retrieved update.

B.2.1 Update initiate message

On receiving an `INITIATE` message the Device must attempt to retrieve the data from the specified URL, and compare the digest of this data with the value specified. If the operation is successful it must send a SRUP Response message (see B.3) — with a status code of `0x00` (`SRUP_UPDATE_SUCCESS`). If the operation fails it must send a `RESPONSE` message with a status code indicating the source of the failure.

- `0xFD` (`SRUP_UPDATE_FAIL_SERVER`) should be sent if the server cannot be reached
- `0xFE` (`SRUP_UPDATE_FAIL_FILE`) should be used if the server cannot supply the file specified
- `0xFF` (`SRUP_UPDATE_FAIL_DIGEST`) should be used if the digest of the retrieved file does not match the value specified in the `INITIATE` message
- Optionally `0xFC` (`SRUP_UPDATE_FAIL_HTTP_ERROR`) may be sent in place of `0xFD` (`SRUP_UPDATE_FAIL_SERVER`) to indicate that a more detailed HTTP error code is available

If the `0xFC` (`SRUP_UPDATE_FAIL_DETAILED`) status code is sent, the Device must then send two SRUP Data messages (see Section B.5) to communicate the HTTP response to the Server. The first `DATA` message must have a Data ID of `HTTP_STATUS` and contain the HTTP status code returned by the web-server in the Data field. The second `DATA` message must have a Data ID of `HTTP_RESPONSE` — and the received HTTP response must be contained within the Data field.

Full details of the `INITIATE` message are shown in Table B.1.

B.2.2 Update activate message

On receiving an `ACTIVATE` message the Device must apply application or system specific procedures to apply an update previously received and specified by the `TOKEN`.

Field	Type	Notes
Message Type	uint8_t	0x01 — SRUP_MESSAGE_TYPE_INITIATE
Version	uint8_t	
Sequence ID	uint64_t	
Sender ID	uint64_t	
Token Length	uint16_t	
Token	uint8_t*	Variable length
Signature Length	uint16_t	
Signature	uint8_t*	Variable length
URL Length	uint16_t	
URL	char*	Variable length
Digest Length	uint16_t	
Digest	uint8_t*	Variable length

TABLE B.1: The SRUP UPDATE INITIATE Message

The Device optionally may then send a further RESPONSE message — signalling the outcome of attempting to apply the update. A STATUS code of 0x10 — SRUP_ACTIVATE_SUCCESS should be used to signal that the activation was successful, and 0x1F — SRUP_ACTIVATE_FAIL should be used if the activation failed.

Full details of the INITIATE message are shown in Table B.2.

Field	Type	Notes
Message Type	uint8_t	0x03 — SRUP_MESSAGE_TYPE_ACTIVATE
Version	uint8_t	
Sequence ID	uint64_t	
Sender ID	uint64_t	
Token Length	uint16_t	
Token	uint8_t*	Variable length
Signature Length	uint16_t	
Signature	uint8_t*	Variable length

TABLE B.2: The SRUP UPDATE ACTIVATE Message

B.3 Response message

The SRUP Response message is used to signal the outcome of a number of operations within SRUP. Permissible status values are defined for each operation type which uses the RESPONSE message.

Full details of the RESPONSE message are shown in Table B.3.

Field	**Type**	**Notes**
Message Type	uint8_t	0x02 — SRUP_MESSAGE_TYPE_RESPONSE
Version	uint8_t	
Sequence ID	uint64_t	
Sender ID	uint64_t	
Token Length	uint16_t	
Token	uint8_t*	Variable length
Signature Length	uint16_t	
Signature	uint8_t*	Variable length
Status	uint8_t	Values: 0x00 — SRUP_UPDATE_SUCCESS 0x10 — SRUP_ACTIVATE_SUCCESS 0x1F — SRUP_ACTIVATE_FAIL 0x20 — SRUP_ACTION_SUCCESS 0x2E — SRUP_ACTION_UNKNOWN 0x2F — SRUP_ACTION_FAIL 0x3F — SRUP_DATA_TYPE_UKNOWN 0x50 — SRUP_JOIN_SUCCESS 0x5E — SRUP_JOIN_REFUSED 0x5F — SRUP_JOIN_FAIL 0x60 — SRUP_OBSERVED_JOIN_VALID 0x6E — SRUP_OBSERVED_JOIN_INVALID 0x6F — SRUP_OBSERVED_JOIN_FAIL 0x70 — SRUP_RESIGN_SUCCESS 0x7F — SRUP_RESIGN_FAIL 0x80 — SRUP_DEREGISTER_SUCCESS 0x8F — SRUP_DEREGISTER_FAIL 0x9F — SRUP_SYNDICATION_END 0xFC — SRUP_UPDATE_FAIL_HTTP_ERROR 0xFD — SRUP_UPDATE_FAIL_SERVER 0xFE — SRUP_UPDATE_FAIL_FILE 0xFF — SRUP_UPDATE_FAIL_DIGEST

TABLE B.3: The SRUP RESPONSE Message

B.4 Action message

The SRUP Action message is used to permit a Server to send an arbitrary command to a Device. The Action ID field denotes the action that is being requested. These values are not defined by the protocol — but are application or system defined values. The recipient may optionally send a RESPONSE message to signal the outcome of the action (0x20 — SRUP_ACTION_SUCCESS or 0x2F — SRUP_ACTION_FAIL).

Actions requiring parametric data can be communicated to the recipient by first sending one or more Data messages containing the parameter(s), before sending the ACTION message.

Messages with an Action ID value unknown to the recipient should be ignored. The recipient may optionally send a RESPONSE message with a status value of 0x2E — SRUP_ACTION_UKNOWN. Full details of the DATA message are shown in Table B.4.

Field	Type	Notes
Message Type	uint8_t	0x04 — SRUP_MESSAGE_TYPE_ACTION
Version	uint8_t	
Sequence ID	uint64_t	
Sender ID	uint64_t	
Token Length	uint16_t	
Token	uint8_t*	Variable length
Signature Length	uint16_t	
Signature	uint8_t*	Variable length
Action ID	uint8_t*	

TABLE B.4: The SRUP ACTION Message

B.5 Data message

The SRUP Data message is designed to permit Servers and Devices to exchange arbitrary data. Each message may consist of a data item, identified by means of the Data ID field. Messages with a Data ID value unknown to the recipient should be ignored. The recipient may send a RESPONSE message with a status value of 0x3F — SRUP_DATA_TYPE_UNKNOWN.

Note that although the values that can be taken by the Data ID field are not defined within the protocol, the values HTTP_ERROR & HTTP_RESPONSE are reserved for use in conjunction with the SRUP_UPADTE_FAIL_DETAILED status of the SRUP_RESPONSE message. (See Section B.2.2), and IDENTIFICATION_RESPONSE is reserved for use in connection with the Identification Request message (See Section B.6).

Full details of the DATA message are shown in Table B.5.

Field	Type	Notes
Message Type	`uint8_t`	`0x05` — `SRUP_MESSAGE_TYPE_DATA`
Version	`uint8_t`	
Sequence ID	`uint64_t`	
Sender ID	`uint64_t`	
Token Length	`uint16_t`	
Token	`uint8_t*`	Variable length
Signature Length	`uint16_t`	
Signature	`uint8_t*`	Variable length
Data ID Length	`uint16_t`	
Data ID	`uint8_t*`	Variable length
Data Length	`uint16_t`	
Data	`uint8_t*`	Variable length

TABLE B.5: The SRUP DATA Message

B.6 Identification request message

There is a requirement for an operator of a device to be able to query that device to identify settings or parameters in use on that device. Whilst the details of these would be application specific, examples might include the specific version or build of the software running on the device, specific hardware version or other details, etc. The Identify Request Message is a message type used to initiate the exchange of Identification information. On receiving the message, the recipient must reply with a `DATA` message (see Section B.5) with the Data ID field containing `IDENTIFICATION_RESPONSE`, and the identification information contained within the Data field.

Full details of the `IDENTIFICATION_REQUEST` message are shown in Table B.6.

Field	Type	Notes
Message Type	`uint8_t`	`0x06` — `SRUP_MESSAGE_TYPE_ID_REQUEST`
Version	`uint8_t`	
Sequence ID	`uint64_t`	
Sender ID	`uint64_t`	
Token Length	`uint16_t`	
Token	`uint8_t*`	Variable length
Signature Length	`uint16_t`	
Signature	`uint8_t*`	Variable length

TABLE B.6: The SRUP IDENTIFY REQUEST Message

B.7 Group messages

An earlier version of the SRUP specification [336], described a series of “group” messages to be used in conjunction with the concept of communications groups (enabling a server to send communications to multiple devices directly).

However due to the requirements to enable secure MQTT messaging (MQTT over TLS) — this has been removed. This eliminates the requirement for a non-standard MQTT broker application to support dynamic access-control lists. C2 Servers may still implement their own virtual groups: but all communication with devices is on a per-device level using a topic corresponding to the device in question.

B.8 Join messages

The Join Message family is used to negotiate Devices joining the control of (and thus becoming subordinate to) a specified C2 Server. A number of different message types are specified in order to provide simple (unmediated) Joining, and both human-mediated, & machine-mediated Joining.

B.8.1 Simple join messages

B.8.1.1 Join request

A Device may send a Join Request message to a given Server requesting to join the C2 network controlled by that Server. Note that unlike other SRUP messages sent from a Device this message must be sent on the MQTT Topic associated with the Server's identity (since, by definition, the Server is not yet subscribed to the Device's topic).

On receiving a Join Request message the Server must either accept the Join — or refuse it.

If accepting it, it must attempt to subscribe to a topic associated with the Sender ID of the `JOIN REQUEST` message, and send a `RESPONSE` message with the Status set to `0x50` — `SRUP_JOIN_SUCCESS` if successful, or a `0x5F` — `SRUP_JOIN_FAIL` if the subscribe fails.

If refusing the request the Server must send a `RESPONSE` message with the Status set to `0x5E` — `SRUP_JOIN_REFUSED`.

Full details of the `JOIN REQUEST` message are shown in Table B.7.

B.8.1.2 Join command

A Server may send a Join Command to a Device: such as in the scenario where the user initiates the Join process by providing the Server with the identity of the Device directly via the Server's user-interface.

Field	Type	Notes
Message Type	`uint8_t`	`0x09` — `SRUP_MESSAGE_TYPE_JOIN_REQ`
Version	`uint8_t`	
Sequence ID	`uint64_t`	
Sender ID	`uint64_t`	
Token Length	`uint16_t`	
Token	`uint8_t*`	Variable length
Signature Length	`uint16_t`	
Signature	`uint8_t*`	Variable length

TABLE B.7: The SRUP JOIN REQUEST Message

On receiving a Join Command message the Device must signal acceptance of this operation by sending a `RESPONSE` message with the Status set to `0x50` — `SRUP_JOIN_SUCCESS`. Since the Server controls to which Devices it sends through the use of the MQTT topic no further action is required by the Device. Optionally the Device may decline the command (subject to system specific implementation) — which may be signalled via a `RESPONSE` message with the Status set to `0x5E` — `SRUP_JOIN_REFUSED`. Note that the Device must respond. The Server must be prepared not to receive a response (such as may be the case if the Device in question is off-line): in which case it should assume that the Join has failed.

Full details of the `JOIN COMMAND` message are shown in Table B.8.

Field	Type	Notes
Message Type	`uint8_t`	`0x0A` — `SRUP_MESSAGE_TYPE_JOIN_COMMAND`
Version	`uint8_t`	
Sequence ID	`uint64_t`	
Sender ID	`uint64_t`	
Token Length	`uint16_t`	
Token	`uint8_t*`	Variable length
Signature Length	`uint16_t`	
Signature	`uint8_t*`	Variable length
Device ID	`uint64_t`	Device ID for device to that is being instructed to become subordinate to the C2 Server

TABLE B.8: The SRUP JOIN COMMAND Message

B.8.2 Human-mediated join messages

B.8.2.1 Human-mediated join request message

If the Server requires additional confirmation of the identity of the Device before accepting the Join, then a human-mediated Join operation can be used. In this scenario the Device initiates the process by sending a Human-Mediated Join Request Message to the Server.

(Note as described in Section B.8.1.1 the Device must send this message using the MQTT topic corresponding to the Server's identity). On receiving this message the Server must then send a Mediated Join Response message to the Device. (See Section B.8.2.2).

Full details of the `HUMAN-MEDIATED JOIN REQUEST` message are shown in Table B.9.

Field	**Type**	**Notes**
Message Type	`uint8_t`	`0x0B` — `SRUP_MESSAGE_TYPE_HM_JOIN_REQ`
Version	`uint8_t`	
Sequence ID	`uint64_t`	
Sender ID	`uint64_t`	
Token Length	`uint16_t`	
Token	`uint8_t*`	Variable length
Signature Length	`uint16_t`	
Signature	`uint8_t*`	Variable length

TABLE B.9: The SRUP HUMAN-MEDIATED JOIN REQUEST Message

B.8.2.2 Human-mediated join response message

The Human-Mediated Join Response Message consists of an encrypted random *confirmation* value sent to the Device (and separately by the C2 system to the user — e.g. over a HTTPS web connection) which can be used by the Device to prove to the human observer that the physical Device presenting the information corresponds to the Device that is requesting the Join operation within SRUP.

In order to practically implement this, the Server must generate a 128-bit random confirmation value, which is then encrypted using the intended recipient's public key (for example using RSA — although this is implementation dependant). This value is then sent within the Human-Mediated Join Response Message. The mechanism for the human observer to check the confirmation value, and then accept / reject is not specified within SRUP.

Full details of the `HUMAN-MEDIATED JOIN RESPONSE` message are shown in Table B.10.

B.8.3 Machine-mediated join messages

In scenarios where no human-observer may be present a machine-mediated version of the Join process is provided. This process requires a trusted third-party (already subordinate to the C2 Server) to take the place of the human observer — with the observation taking-place over a short-range point-to-point communication channel outside of SRUP. The protocol does not specify the mechanism for this observation to take place.

Field	Type	Notes
Message Type	uint8_t	0x0C — SRUP_MESSAGE_TYPE_HM_JOIN_RESP
Version	uint8_t	
Sequence ID	uint64_t	
Sender ID	uint64_t	
Token Length	uint16_t	
Token	uint8_t*	Variable length
Signature Length	uint16_t	
Signature	uint8_t*	Variable length
Conf. Length	uint16_t	
Encrypted Conf.	uint8_t*	Length will be RSA key-length / 8 e.g. 256 bytes for a 2048-bit key. Unencrypted confirmation value is 128-bits implemented as uint8_t[8]

TABLE B.10: The SRUP HUMAN-MEDIATED JOIN RESPONSE Message

B.8.3.1 Observed join request message

The Device must first identify details of the C2 Server and the Observer outside of the protocol. Once this has been done, the Device must send a Observed Join Request Message to the Server. (Note as described in Section B.8.1.1 the Device must send this message using the MQTT topic corresponding to the Server's identity).

The Observed Join Request Message contains the identity of the Observer node in the Observer ID field.

On receiving a valid Observed Join Request message the Server must then send an Observed Join Response message back to the Device, and an Observation Request message to the Observer Node. (See Sections B.8.3.2 & B.8.3.3).

Full details of the `OBSERVED JOIN REQUEST` message are shown in Table B.11.

Field	Type	Notes
Message Type	uint8_t	0x0D — SRUP_MESSAGE_TYPE_OBS_JOIN_REQ
Version	uint8_t	
Sequence ID	uint64_t	
Sender ID	uint64_t	
Token Length	uint16_t	
Token	uint8_t*	Variable length
Signature Length	uint16_t	
Signature	uint8_t*	Variable length
Observer ID	uint64_t	

TABLE B.11: The SRUP OBSERVED JOIN REQUEST Message

B.8.3.2 Observed join response message

The Observed Join Response message is identical in content to the Human-Mediated Join Response Message (Section B.8.2.1), however it is included as a discrete message type to simplify the implementation of systems in which both human- and machine-mediated join operations may occur.

Upon receiving a Observed Join Response Message a Device must transmit the unencrypted Confirmation value to the observer externally to the protocol.

Full details of the `OBSERVED JOIN RESPONSE` message are shown in Table B.12.

Field	Type	Notes
Message Type	`uint8_t`	`0x0E` — `SRUP_MESSAGE_TYPE_OBS_JOIN_RESP`
Version	`uint8_t`	
Sequence ID	`uint64_t`	
Sender ID	`uint64_t`	
Token Length	`uint16_t`	
Token	`uint8_t*`	Variable length
Signature Length	`uint16_t`	
Signature	`uint8_t*`	Variable length
Conf. Length	`uint16_t`	
Encrypted Conf.	`uint8_t*`	Length will be RSA key-length / 8 e.g. 256 bytes for a 2048-bit key. Unencrypted confirmation value is 128-bits implemented as `uint8_t[8]`

TABLE B.12: The SRUP OBSERVED JOIN RESPONSE Message

B.8.3.3 Observation request message

The Observation Request Message is used to communicate the Confirmation value to the Observer node. The format is similar to the `OBSERVED JOIN RESPONSE` Message (Section B.8.3.2) — though the Confirmation value is encrypted using the Observation Node's public key. It also contains an additional field — to be used to store the device ID of the *joining device*. This is required to permit the observer to respond to the C2 Server, as which device it has observed, in the event that multiple observations are requested within the same time period.

On receiving an `OBSERVATION REQUEST` message the Observer Node must prepare to receive the Confirmation value from the Device requesting the Join, externally to the protocol, and then to compare the received value to the value contained within the `OBSERVATION REQUEST` message. The Observer must then send a `RESPONSE` message back to the Server to signal the outcome of the comparison. This `RESPONSE` message should have a Status of `0x60` — `SRUP_OBSERVED_JOIN_VALID` if the two values match, or a

Status of `0x6E` — `SRUP_OBSERVED_JOIN_INVALID` if they do not. A Status of `0x6F` — `SRUP_OBSERVED_JOIN_FAIL` may be used if the operation fails (such as no data is received by the Observer).

Full details of the `OBSERVATION REQUEST` message are shown in Table B.12.

Field	Type	Notes
Message Type	`uint8_t`	`0x0F` — `SRUP_MESSAGE_TYPE_OBSERVE_REQ`
Version	`uint8_t`	
Sequence ID	`uint64_t`	
Sender ID	`uint64_t`	
Token Length	`uint16_t`	
Token	`uint8_t*`	Variable length
Signature Length	`uint16_t`	
Signature	`uint8_t*`	Variable length
Joining Device ID	`uint64_t`	
Conf. Length	`uint16_t`	
Encrypted Conf.	`uint8_t*`	Length will be RSA key-length / 8 e.g. 256 bytes for a 2048-bit key. Unencrypted confirmation value is 128-bits implemented as `uint8_t[8]`

TABLE B.13: The SRUP OBSERVATION REQUEST Message

B.9 Resignation and termination messages

In order to remove a Device from the control of a Server, SRUP provides Resignation & Termination message types.

B.9.1 Resign request

If a Device wishes to resign from the control of a Server it may send a Resign Request Message to the Server. Upon receiving a `RESIGN REQUEST` message the Server may either accept or reject the request, and then it must send a `RESPONSE` message — with a Status of `0x70` — `SRUP_RESIGN_SUCCESS`, or `0x7F` — `SRUP_RESIGN_FAIL` if the resignation request is not accepted.

Full details of the `RESIGN REQUEST` message are shown in Table B.14.

B.9.2 Termination command

A Server may send a Termination Command message to any subordinate Device to instruct it that it is no longer subject to control by the Server. On receiving a Termination

Field	Type	Notes
Message Type	`uint8_t`	`0x10` — `SRUP_MESSAGE_TYPE_RESIGN_REQ`
Version	`uint8_t`	
Sequence ID	`uint64_t`	
Sender ID	`uint64_t`	
Token Length	`uint16_t`	
Token	`uint8_t*`	Variable length
Signature Length	`uint16_t`	
Signature	`uint8_t*`	Variable length

TABLE B.14: The RESIGN REQUEST Message

Command message the Device may optionally send a `RESPONSE` message — with a Status of `0x70` — `SRUP_RESIGN_SUCCESS` to the Server (via the MQTT topic associated with the Server's identity). The Server must not send any further messages to the Device without another Join operation (see Section B.8) first taking place.

Full details of the `TERMINATION COMMAND` message are shown in Table B.15.

Field	Type	Notes
Message Type	`uint8_t`	`0x11` — `SRUP_MESSAGE_TYPE_TERMINATE_COM`
Version	`uint8_t`	
Sequence ID	`uint64_t`	
Sender ID	`uint64_t`	
Token Length	`uint16_t`	
Token	`uint8_t*`	Variable length
Signature Length	`uint16_t`	
Signature	`uint8_t*`	Variable length

TABLE B.15: The TERMINATION COMMAND Message

B.10 Deregistration messages

Devices may be permanently removed from the system via a Deregistration message. Deregistration causes the Device's public key to be deleted from the System. Devices should also remove the Server public keys that they hold, upon notification of deregistration.

B.10.1 Deregister request

A Device wishing to be permanently deregistered from a given system may send a Deregister Request message to any Server to which it is subordinate. The Server must then send a final `RESPONSE` message to the Device with a status of `0x80` — `SRUP_DEREGISTER_SUCCESS`, or in the event that the Deregistration cannot be processed, a status of `0x8F` — `SRUP_DEREGISTER_FAIL` may be sent. This should be used exceptionally however as Deregistration requests should always be honoured unless they cannot be processed. The Device must participate in Registration (see Section B.11) and Join (Section B.8) operations before it is able to receive further SRUP messages.

Full details of the `DEREGISTER REQUEST` message are shown in Table B.15.

Field	Type	Notes
Message Type	`uint8_t`	`0x12` — `SRUP_MESSAGE_TYPE_DEREGISTER_REG`
Version	`uint8_t`	
Sequence ID	`uint64_t`	
Sender ID	`uint64_t`	
Token Length	`uint16_t`	
Token	`uint8_t*`	Variable length
Signature Length	`uint16_t`	
Signature	`uint8_t*`	Variable length

TABLE B.16: The DEREGISTER REQUEST Message

B.10.2 Deregister command

A Server may send a Deregister Command message to any subordinate Device to instruct it that it is no longer Registered within the system and that its access has been revoked. A Device receiving a Deregister Command should not attempt to send any further SRUP messages — and it should disconnect from the MQTT Broker being used for SRUP messages. Full details of the `DEREGISTER COMMAND` message are shown in Table B.17.

Field	Type	Notes
Message Type	`uint8_t`	`0x13` — `SRUP_MESSAGE_TYPE_DEREGISTER_COM`
Version	`uint8_t`	
Sequence ID	`uint64_t`	
Sender ID	`uint64_t`	
Token Length	`uint16_t`	
Token	`uint8_t*`	Variable length
Signature Length	`uint16_t`	
Signature	`uint8_t*`	Variable length

TABLE B.17: The DEREGISTER COMMAND Message

B.11 Registration

The process of a Device performing initial Registration to facilitate registration of the Device's public key is performed entirely outside of the SRUP protocol. An explanation of the process is included for completeness — and an outline of a reference implementation is described.

B.11.1 Registration requirements

SRUP requires that key exchange has taken place before the first Join operation. It is therefore assumed that the Device will already have a copy of the public key for any Server that it wishes to communicate with (or for any which would wish to communicate with it), and that all Servers have a copy of the public key for any Devices that they will communicate before SRUP operations can commence.

B.11.2 Example reference registration scheme

It is expected that implementations will use an HTTPS web API running over TLS. Using this approach the Device would send its Identity and public key to the web-server via a HTTPS `POST` request (together with any additional system implementation specific information required), and receive a response from the web-service containing the address of the MQTT broker and the Server public key (or a HTTPS URL from which the key or keys may be retrieved — e.g. by using a `GET` request against an end-point corresponding to the Server ID, though again this is on an implementation-specific basis).

Once registration is complete the C2 server(s) need to receive a copy of the keys. Similarly a mechanism should be provided for a Server to, on receipt of a Deregister Request (See Section B.10.1), propagate the requirement for removal of the key to all other Servers.

B.12 Syndication messages

Syndication messages were introduced to address the requirement to provide shared access from one C2 system to another without a fully trusted relationship between the two. The syndication family of messages consists of the following message types:

- Syndication Init
- Syndication Request
- Syndicated Device Count
- Syndicated Device List
- Syndicated Data
- Syndicated Action
- Syndicated IQ Request
- Syndicated C2 Request

- Syndicated End Request
- Syndicated Terminate Command

There are three primary parties involved in syndication operations.

- The *Syndicating Server* is the C2 server that is requesting to receive data from another C2 system
- The *Syndicated Server* is the C2 server that is sharing its data
- The *Syndication Device* is a special class of SRUP device which acts as a *bridge* between the two SRUP universes involved in the syndication

B.12.1 Syndication initialization

The `SYNDICATION INIT` message is used to initialize syndication operations. It is sent from the syndicating server to the syndication device, and contains all of the data that is required by the syndication device to initialize syndication operations — including registering an identity in the syndicated server's universe.

Full details of the `SYNDICATION INIT` message are shown in Table B.18.

Field	**Type**	**Notes**
Message Type	`uint8_t`	`0x21` — `SRUP_MESSAGE_TYPE_SYNDICATION_INIT`
Version	`uint8_t`	
Sequence ID	`uint64_t`	
Sender ID	`uint64_t`	
Token Length	`uint16_t`	
Token	`uint8_t*`	Variable length
Signature Length	`uint16_t`	
Signature	`uint8_t*`	Variable length
URL Length	`uint16_t*`	
Registration URL	`char*`	Variable length
Key Length	`uint16_t`	
Syndication Key	`uint8_t`	Variable length

TABLE B.18: The SRUP SYNDICATED INIT Message

B.12.2 Syndication request

Once a syndication device has registered and joined a syndicated C2 server — the syndication device will then send a `SYNDICATION REQUEST` message to request the syndication is commenced. This will include the key value used to authenticate the request.

Details of the `SYNDICATION INIT` message are shown in Table B.19.

Field	Type	Notes
Message Type	`uint8_t`	`0x29` — `SRUP_MESSAGE_TYPE_SYNDICATION_REQUEST`
Version	`uint8_t`	
Sequence ID	`uint64_t`	
Sender ID	`uint64_t`	
Token Length	`uint16_t`	
Token	`uint8_t*`	Variable length
Signature Length	`uint16_t`	
Signature	`uint8_t*`	Variable length
Key Length	`uint16_t`	
Syndication Key	`uint8_t`	Variable length

TABLE B.19: The SRUP SYNDICATION REQUEST Message

B.12.3 Syndicated device count

Having received a valid `SYNDICATION REQUEST` message, the syndicated server must then respond by sending a `SYNDICATED DEVICE COUNT` message to the syndication device, which will then send a second `SYNDICATED DEVICE COUNT` message to the syndicating server. This will contain the total number of devices that the syndicated server is going to share with the syndicating server.

The details of the `SYNDICATED DEVICE COUNT` message are shown in Table B.20.

Field	Type	Notes
Message Type	`uint8_t`	`0x27` — `SRUP_MESSAGE_TYPE_SYNDICATED_DEV_COUNT`
Version	`uint8_t`	
Sequence ID	`uint64_t`	
Sender ID	`uint64_t`	
Token Length	`uint16_t`	
Token	`uint8_t*`	Variable length
Signature Length	`uint16_t`	
Signature	`uint8_t*`	Variable length
Device Sequence	`uint32_t`	
Device ID	`uint64_t`	

TABLE B.20: The SRUP SYNDICATED DEVICE COUNT Message

B.12.4 Syndicated device list

Having sent the `SYNDICATED DEVICE COUNT` message, the syndicated server will then send a series of `SYNDICATED DEVICE LIST` messages, consisting of the ID for a given device from the set of devices it is sharing with the syndicating server, and an ordinal index value for that device. Again this is initially sent to the syndication device, which then sends a copy on to the syndicating server.

The details of the `SYNDICATED DEVICE LIST` message are shown in Table B.21.

Field	**Type**	**Notes**
Message Type	`uint8_t`	`0x28` — `SRUP_MESSAGE_TYPE_SYNDICATED_DEV_LIST`
Version	`uint8_t`	
Sequence ID	`uint64_t`	
Sender ID	`uint64_t`	
Token Length	`uint16_t`	
Token	`uint8_t*`	Variable length
Signature Length	`uint16_t`	
Signature	`uint8_t*`	Variable length
Device Count	`uint32_t`	

TABLE B.21: The SRUP SYNDICATED DEVICE LIST Message

B.12.5 Syndicated data

The `SYNDICATED DATA` message is used to share a data value with a syndicating server. The message is identical to the regular `DATA` message — with the addition of a 64-bit Source ID field appended to the end of the message. This is used to denote the ID of the *source device* from where the original data message originated. The `SYNDICATED DATA` message is sent by the syndicated server (and never by the source device itself).

Full details of the `SYNDICATED DATA` message are shown in Table B.22.

Field	**Type**	**Notes**
Message Type	`uint8_t`	`0x25` — `SRUP_MESSAGE_TYPE_SYNDICATED_DATA`
Version	`uint8_t`	
Sequence ID	`uint64_t`	
Sender ID	`uint64_t`	
Token Length	`uint16_t`	
Token	`uint8_t*`	Variable length
Signature Length	`uint16_t`	
Signature	`uint8_t*`	Variable length
Data ID Length	`uint16_t`	
Data ID	`uint8_t*`	Variable length
Data Length	`uint16_t`	
Data	`uint8_t*`	Variable length
Source ID	`uint64_t`	

TABLE B.22: The SRUP SYNDICATED DATA Message

B.13 Syndicated action

The `SYNDICATED ACTION` message is analogous to the regular SRUP `ACTION` message, and is used to send a request for an action to be performed, from a syndicating C2 server to a syndicated server. As with all other syndication messages the SYNDICATED ACTION message is sent via the syndication device.

Details of the SYNDICATED ACTION message are shown in Table B.23.

Field	Type	Notes
Message Type	uint8_t	0x24 — SRUP_MESSAGE_TYPE_SYNDICATED_ACTION
Version	uint8_t	
Sequence ID	uint64_t	
Sender ID	uint64_t	
Token Length	uint16_t	
Token	uint8_t*	Variable length
Signature Length	uint16_t	
Signature	uint8_t*	Variable length
Action ID	uint8_t*	
Source ID	uint64_t	

TABLE B.23: The SRUP SYNDICATED ACTION Message

B.14 Syndicated ID request

The SYNDICATED ID REQUEST message is used to allow a syndicating server to send a request for the ID string for a given device. This is optional but may typically be sent after reception of the SYNDICATED DEVICE LIST message corresponding to that device. On receiving the SYNDICATED ID REQUEST message from a syndication device, a syndicated server must either send the current ID value it has previously received from that device, or send a ID REQUEST message to the device in order to update the value. The response is sent to the requester as a SYNDICATED DATA message, using the IDENTIFICATION_RESPONSE data ID. The device for which the ID is being made is included in the Target ID field.

The details of the SYNDICATED ID REQUEST message are shown in Table B.24.

Field	Type	Notes
Message Type	uint8_t	0x26 — SRUP_MESSAGE_TYPE_SYNDICATED_ID_REQUEST
Version	uint8_t	
Sequence ID	uint64_t	
Sender ID	uint64_t	
Token Length	uint16_t	
Token	uint8_t*	Variable length
Signature Length	uint16_t	
Signature	uint8_t*	Variable length
Target ID	uint64_t	

TABLE B.24: The SRUP SYNDICATED ID REQUEST Message

B.14.1 Syndicated C2 request

The SYNDICATED C2 REQUEST message is used to send a request from a syndicating server to a syndicated server requesting that the syndicated server performs one of a (previously determined and agreed) set of C2 operations. These are not defined within the specification, but could include requesting an update operation, or the change of system configuration. The message consists of a *standard* SRUP message, plus a 256-bit integer to denote which of the operations is being requested, and a byte-stream consisting of any required configuration data.

The details of the SYNDICATED C2 REQUEST message are shown in Table B.25.

Field	**Type**	**Notes**
Message Type	uint8_t	0x2C — SRUP_MESSAGE_TYPE_SYNDICATED_C2_REQ
Version	uint8_t	
Sequence ID	uint64_t	
Sender ID	uint64_t	
Token Length	uint16_t	
Token	uint8_t*	Variable length
Signature Length	uint16_t	
Signature	uint8_t*	Variable length
Data Length	uint16_t	
Data	uint8_t*	Variable length
Request ID	uint8_t	

TABLE B.25: The SRUP SYNDICATED C2 REQUEST Message

B.14.2 Syndicated end request and syndicated termination

There are two ways that syndication operations can be ended. Either the syndicating server may send a SYNDICATION END REQUEST message to the syndicated server, to request that no further syndication messages are sent, or the syndicated server may send a SYNDICATION TERMINATION message to inform the syndicating server than syndication operations have been ended by the remote party.

On receipt of a SYNDICATION END REQUEST message the syndicated server should respond with a RESPONSE message with a status of SRUP_SYNDICATION_END (0x9F).

The full details of the SYNDICATION TERMINATION message and the SYNDICATION END REQUEST message are shown in Figures B.26 & B.27.

Field	Type	Notes
Message Type	uint8_t	0x2F — SRUP_MESSAGE_TYPE_SYNDICATION_TERMINATE
Version	uint8_t	
Sequence ID	uint64_t	
Sender ID	uint64_t	
Token Length	uint16_t	
Token	uint8_t*	Variable length
Signature Length	uint16_t	
Signature	uint8_t*	Variable length

TABLE B.26: The SRUP SYNDICATION TERMINATION Message

Field	Type	Notes
Message Type	uint8_t	0x2E — SRUP_MESSAGE_TYPE_SYNDICATION_END
Version	uint8_t	
Sequence ID	uint64_t	
Sender ID	uint64_t	
Token Length	uint16_t	
Token	uint8_t*	Variable length
Signature Length	uint16_t	
Signature	uint8_t*	Variable length

TABLE B.27: The SRUP SYNDICATION END REQUEST Message

Appendix C

Timing Experiment

In order to determine the performance of the cryptographic algorithms supporting the SRUP protocol on low-cost commodity hardware, their performance was measured on a Raspberry Pi 3.

This experiment was run on 16th September 2017.

A program built from the source code shown in Source Code listing C.1 was run to measure the time taken for the sign & verify functions to execute.

```
#include <SRUP.h>
#include <SRUP_Init.h>

#include <chrono>
#include <iostream>

#define TARGET "TARGET"
#define TOKEN  "e0cfa165-507e-469c-9163-170322ece413"
#define URL    "http://iot-lab.uk/SRUP.device.py"
#define DIGEST "6c9d09233f0ead4fc3824af2752423dda352e979cf3e356dc4b6f2bafb14cbd"
#define PVKEY  "private_key.pem"
#define PBKEY  "public_key.pem"

int main()
{
    char* target;
    char* token;
    char* url;
    char* digest;
    char* pvkeyfile;
    char* pbkeyfile;

    unsigned char* r_serial_data;
    unsigned char* s_serial_data;
    int sz;

    SRUP_MSG_INIT *msg_init;
    SRUP_MSG_INIT *msg_init2;

    // Setup...
    msg_init = new SRUP_MSG_INIT;
    target = new char[std::strlen(TARGET)];
    std::strcpy(target, TARGET);
    token = new char[std::strlen(TOKEN)];
    std::strcpy(token, TOKEN);
    url = new char[std::strlen(URL)];
    std::strcpy(url, URL);
    digest = new char[std::strlen(DIGEST)];
    std::strcpy(digest, DIGEST);
    pvkeyfile = new char[std::strlen(PVKEY)];
    std::strcpy(pvkeyfile, PVKEY);
    pbkeyfile = new char[std::strlen(PBKEY)];
    std::strcpy(pbkeyfile, PBKEY);

    msg_init->token(token);
    msg_init->target(target);
    msg_init->url(url);
    msg_init->digest(digest);

    // Time to execute "sign"
    auto t0 = std::chrono::high_resolution_clock::now();
    msg_init->Sign(pvkeyfile);
    auto t1 = std::chrono::high_resolution_clock::now();
    auto dts = 1.e-9 * std::chrono::duration_cast<std::chrono::nanoseconds>(t1-↙
    ↳ t0).count();

    std::cout << "'SIGN' Time: " << dts << " seconds." << std::endl;

    // Setup for verify...
    r_serial_data = msg_init->Serialized();
    sz = msg_init->SerializedLength();

    msg_init2 = new SRUP_MSG_INIT;
    s_serial_data = new unsigned char[sz];
    std::memcpy(s_serial_data, r_serial_data, sz);
    msg_init2->DeSerialize(s_serial_data);

    // Time to execute "verify"
    auto t2 = std::chrono::high_resolution_clock::now();
    msg_init2->Verify(pbkeyfile);
    auto t3 = std::chrono::high_resolution_clock::now();
    auto dtv = 1.e-9 * std::chrono::duration_cast<std::chrono::nanoseconds>(t3-
    ↳ t2).count();

    std::cout << "'VERIFY' Time: " << dtv << " seconds." << std::endl;

    // Clean-up
    delete(msg_init);
    delete(target);
    delete(token);
    delete(url);
    delete(digest);
    delete(pvkeyfile);
    delete(pbkeyfile);
    delete(msg_init2);
    delete(s_serial_data);

    return 0;
}
```

LISTING C.1: A C++ test program to be used to measure the performance of the SRUP sign and verify functions when running on a Raspberry Pi

This code was compiled, using default compiler flags, and the resulting executable was run 5 times, with the output captured from `stdout` to a series of text files. The raw data is shown here.

```
'SIGN' Time: 0.0619318 seconds.
'VERIFY' Time: 0.0107816 seconds.

'SIGN' Time: 0.0616033 seconds.
'VERIFY' Time: 0.0104205 seconds.

'SIGN' Time: 0.0368569 seconds.
'VERIFY' Time: 0.00643931 seconds.

'SIGN' Time: 0.0617651 seconds.
'VERIFY' Time: 0.011163 seconds.

'SIGN' Time: 0.0612372 seconds.
'VERIFY' Time: 0.0107523 seconds.
```

	SIGN	VERIFY
Test 1	61.93 ms	10.78 ms
Test 2	61.60 ms	10.42 ms
Test 3	36.86 ms	6.439 ms
Test 4	61.77 ms	11.16 ms
Test 5	61.24 ms	10.75 ms
Arithmetic Mean	56.67 ms	9.911 ms

TABLE C.1: Execution time, in milliseconds, of the SRUP cryptographic functions on Raspberry Pi 3B hardware

The calculations were then performed using a Microsoft Excel spreadsheet, the summary results from which are shown in Table C.1.

The full dataset for this experiment is available from the University of Southampton repository at https://doi.org/10.5258/SOTON/D0486 [320].

Appendix D

Performance Comparison Experiment — device code

This Appendix contains a listing of the code used for the SRUP performance experiments described in Chapter 11.

D.1 SRUP device code

```python
from gpiozero import LED, Button
import time
import pySRUP
from datetime import datetime
import logging
import logging.handlers
import atexit
import coloredlogs

running = True
led_state = False
ready = False
button=Button(17, False)
orange_led = LED(22)
blue_led = LED(27)

def make_ready():
    global ready
    ready = True

def resign(srup_client):
    old_server_id = srup_client.server_id
    srup_client.send_SRUP_Resign_Request()
    # To avoid having to reset the demo every-time
    # we'll restore the server for the config file..
    srup_client.server_id = old_server_id
    logging.info("Saving settings...")
    client.save_settings()
    logging.info("Exiting client_demo.")
    global running
    running = False

def on_action(msg_action):
    global led_state
    if msg_action.action_id == 0x00:
        now = datetime.now()
        blue_led.on()
        logging.info("ACTION\tSTART\t{}".format(now))
    elif msg_action.action_id == 0xFF:
        now = datetime.now()
        blue_led.off()
        logging.info("ACTION\tSTOP\t{}".format(now))

def on_join_succeed():
    logging.info("Join Accepted")
    orange_led.on()

def on_terminate():
    global running
    orange_led.off()
    client.save_settings()
    logging.info("Termination command received - leaving C2")
    running = False

LOG_FILE = "ex1_D1.log"

logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
log_format_string = '%(asctime)s.%(msecs)03d \t [%(levelname)s] \t %(message)s'

fHandler = logging.FileHandler(LOG_FILE)
fHandler.setLevel(logging.INFO)
f_format = logging.Formatter(log_format_string, "%Y-%m-%d %H:%M:%S")
fHandler.setFormatter(f_format)
logger.addHandler(fHandler)

coloredlogs.DEFAULT_LOG_FORMAT = log_format_string
coloredlogs.install(level='INFO')

```

```
button.when_pressed = make_ready

client = pySRUP.Client("ex1_dev.cfg", "https://beta.local", "web.crt", "TEST")
client.on_action(on_action)
client.on_join_succeed(on_join_succeed)
client.on_terminate(on_terminate)

logging.info("Device {} starting up...".format(client.id))

while not ready:
    orange_led.on()
    time.sleep(0.25)
    orange_led.off()
    time.sleep(0.25)

logging.info("Done waiting")

with client:
    # Start by joining the server defined in the config...
    client.send_SRUP_simple_join()
    try:
        while running:
            pass

    except KeyboardInterrupt:
        logging.info("User requested exit - via Keyboard Interrupt...")
        resign(client)
```

LISTING D.1: SRUP Device Code

D.2 MQTT device code

```python
from gpiozero import LED, Button
import paho.mqtt.client as mqtt
import time
from datetime import datetime
import logging
import logging.handlers
import coloredlogs

running = True
led_state = False
ready = False
button=Button(17, False)
orange_led = LED(22)
blue_led = LED(27)

dev_id = 1

# The callback for when the client receives a CONNACK response from the server.
def on_connect(client, userdata, flags, rc):
    # Subscribing in on_connect() means that if we lose the connection and
    # reconnect then subscriptions will be renewed.
    client.subscribe("test/d{}".format(dev_id), qos=1)

# The callback for when a PUBLISH message is received from the server.
def on_message(client, userdata, msg):
    # print(msg.payload)
    global led_state
    if msg.payload.decode() == '1':
        now = datetime.now()
        blue_led.on()
        logging.info("ACTION\tSTART\t{}".format(now))

    elif msg.payload.decode() == '0':
        now = datetime.now()
        blue_led.off()
        logging.info("ACTION\tSTOP\t{}".format(now))

    elif msg.payload.decode() == 'X':
        terminate()

def make_ready():
    global ready
    ready = True

def terminate():
    global running
    orange_led.off()
    logging.info("Terminate recieved - exiting")
    running = False

LOG_FILE = "ex1_mqtt_d{}.log".format(dev_id)

logger = logging.getLogger()
logger.setLevel(logging.DEBUG)

log_format_string = '%(asctime)s.%(msecs)03d \t [%(levelname)s] \t %(message)s'

fHandler = logging.FileHandler(LOG_FILE)
fHandler.setLevel(logging.INFO)

f_format = logging.Formatter(log_format_string, "%Y-%m-%d %H:%M:%S")
fHandler.setFormatter(f_format)

logger.addHandler(fHandler)

coloredlogs.DEFAULT_LOG_FORMAT = log_format_string
coloredlogs.install(level='INFO')

client = mqtt.Client(client_id="Device {} Client".format(dev_id))
client.on_connect = on_connect
client.on_message = on_message

client.connect("beta.local",1883, 60)
button.when_pressed = make_ready

logging.info("Device {} starting up...".format(dev_id))

while not ready:
    orange_led.on()
    time.sleep(0.25)
    orange_led.off()
    time.sleep(0.25)

orange_led.on()
client.publish("test", payload=dev_id, qos=1)
client.loop()
logging.info("Done waiting")

try:
    while running:
        client.loop()

```

```
except KeyboardInterrupt:
    logging.info("User requested exit - via Keyboard Interrupt...")

finally:
    client.disconnect()
```

LISTING D.2: MQTT Device Code

Appendix E

Network Conditioning Simulation Setup

During the experimental work (11), the following network conditioning parameters were used. Experiment 1 used no network-conditioning. Experiments 2–5 used network delay distribution data tables taken from [324]. The specific delay data table used is described here. For experiments 6–10, `tc` settings were directly applied (based on data from [323], as well as published standards and observations made). The `tcset` tool was used to load these parameters, and the parameters used are shown in Table E.1.

Experiment	Description	Distribution Used	*tc* Settings Applied
1	LAN Ethernet	No network conditioning	-
2	Good Strength 4G	`H3G_Access_AB.good.4G.no_roaming`	-
3	Medium Strength 4G	`TIM.medium.4G.no_roaming`	-
4	Good 3G	`TIM.good.3G.no_roaming`	
5	Poor 3G	`TIM.bad.3G.no_roaming`	
6	2G EDGE best-case	-	`−rate 384Kbps −delay 115.6ms −delay-distro 33.6`
7	Observed 3G poor signal	-	`−rate 115Kbps −delay 122.6ms −delay-distro 40.6`
8	2G GPRS best-case	-	`−rate 40Kbps −delay 122.6ms −delay-distro 40.6`
9	3G poor signal + 5% loss	-	`−rate 115Kbps −delay 122.6ms −delay-distro 40.6 −loss 5`
10	2G GPRS + 10% loss	-	`−rate 40Kbps −delay 122.6ms −delay-distro 40.6 −loss 10`

TABLE E.1: The detailed network conditioning settings for each of the ten experiments.

Appendix F

Security Analysis of the Secure Remote Update Protocol

This Appendix contains a more detailed description of the analysis and assessment of the security of the SRUP protocol and its software implementation, as described elsewhere within this thesis.

F.1 Secure by Design

Although pre-dating the 2018 DCMS report "*Secure by Design: Improving the cyber security of consumer Internet of Things*" [337], and the subsequent code of practice [125], the SRUP protocol was designed from the outset to be secure, meeting the design principles of the DCMS code of practice (see Chapter 8). The design incorporated features in order to be robust against attacks versus the protocol itself, and against attacks targeting the protocol's associated infrastructure.

F.1.1 Adversarial Design

The primary consideration when the protocol was designed was to preemptively counter potential attacks against it. This was accomplished by taking an adversarial approach to the design, specifically: *"If I had hostile intent, how would I attack this? How can I break it? How can I access the data? What would I need to do to be able to interfere with it?"*. Countermeasures against these attacks were then identified and incorporated into the design from the outset, long before any code had been written. This approach was also taken for each subsequent feature being added.

F.1.2 Use of extant and trusted libraries

Another significant way that SRUP ensures correctness of operation is by making extensive use of extant library code. Wherever possible in the development of the SRUP code-base, widely-used — and therefore likely already well-tested and validated — libraries were utilized. In all cases, the library to be used was selected on the basis of it being the ‘best in class’ for the specific requirement: determined by factors such as how frequently used and frequently updated the library was, as well as on the basis of feature selection. As described in Section 2.5, widely-used libraries will be more likely to have received significant amounts of scrutiny and examination

Using an extant library is particularly important in the context of cryptography, since developing and ensuring the correctness of cryptographic routines is especially challenging. However, in addition to using ostensibly correct *implementations* of cryptographic functions via `libcrypto`, it is also essential to ensure that these functions are *used* correctly. To this end, all routines where cryptographic functions are called are written based on (and are consistent with) code examples within the the OpenSSL documentation [22]. This ensures that, for example, *RSA envelope padding* is correctly implemented using the provided library functions in the correct combination.

F.1.3 Library code used within SRUP

As described in Chapter 9, the code for the SRUP protocol itself comprises three main parts:

- The underpinning C++ library — `libSRUP`
- The Python wrapper of this library — `pySRUP_lib`
- The object-oriented Python package — `pySRUP`

The fourth and final element, consists of the example implementations of C2 server and other back-end services, as well as the example implementations of devices, etc.

The library code used by each of these four parts is summarized in Table F.1.

Note: although the example implementations of the back-end services make use of `flask` to provide HTTP & HTTPS server-side web application capabilities, the actual web serving is implemented in the example using `nginx` running within a docker container (see Section 9.6.2).

SRUP Component	Language	Library Used	Purpose
libSRUP	C++	OpenSSL (`libcrypto.a`)	Cryptographic functions
pySRUP_Lib	C++	Boost Python	Enable wrapping of the C++ class for access as a native binary CPython library
pySRUP	Python	`paho.mqtt.client`	MQTT client implementation
	Python	`requests`	HTTP and HTTPS client implementation
	Python	`cryptography`	Implementations of RSA and X.509
SRUP Backend	Python	`flask`	HTTP & HTTPS server-side web application capabilities

TABLE F.1: The extant libraries used by different components of the SRUP system.

F.2 MQTT Security Assessment

Independent of this research, an (unpublished) assessment of the potential vulnerabilities in the MQTT protocol in the context of IoT was undertaken by the Defence Science and Technology Laboratory (Dstl) in 2017. This concluded, in part, that some of the major issues with the use of MQTT were its lack of encryption (by default), the lack of message integrity, and the lack of an authenticated way to identify the sender of a message. Although predating that assessment, the development of the SRUP protocol addresses each of these points, mitigating the weaknesses in MQTT highlighted.

F.3 Static and Dynamic Analysis

All of the code written was subjected to extensive static and dynamic analysis to ensure validity. The static analysis of the C++ code was conducted using the *CppCheck* tool [338] built into the *CLion* Integrated Development Environment (IDE). For dynamic analysis, Valgrind tools [339] were used to identify errors such as memory leaks.

F.3.1 Static Analysis

All compiler warnings having previously been addressed, the code was subjected to analysis by CppCheck in order to identify issues such as incorrectly initialized constructors, incorrectly freed dynamic memory, or other structural errors. All such issues were then rectified, either using the recommendations from CppCheck directly or by otherwise re-writing the code section in question.

In particular running CppCheck on the SRUP codebase, generated two main types of recommendations.

The first of these was to mark as `const`, a number of *getter* methods for class properties which return, unchanged, a protected member variable (see an example of this in Listing F.1).

```
const uint64_t *SRUP_MSG::senderID()
{
    return m_sender_ID;
}
```

LISTING F.1: Example of a `const` property *getter* function

The second recommendation was to change the way that the `delete` function is used in a few places where it didn't correctly match the way that the heap memory had been allocated. When `delete(ptr1)` is called, it will free the memory at the location being pointed to by `ptr1`; but if `ptr1` is actually pointing to an array (e.g. the memory in question was allocated using `new uint8_t[128];`) then in order to free the whole structure (and not just the initial location), `delete` should be called using the syntax `delete[] ptr1;`

See Listings F.2, F.3 & F.4 for examples of this being used correctly: depicting the constructor, the setter method for a property, and the class destructor.

F.3.2 Dynamic Analysis

Execution of the C++ binary library was subsequently examined using Valgrind's *memcheck* tool, to attempt to identify any run-time memory leaks or invalid heap pointer accesses. In all cases any identified issues were rectified.

F.3.3 Analysis of Python Code

The Python code was also subjected to static analysis using the *PyLint* tool [340] to ensure that it met the standard of PEP8 [341]. Although this is not a guarantee of correctness, it does check adherence to the very widely-adopted Python coding standard. Failure to adhere to standards is one source of 'code smells' [342] and can easily conceal errors. As before, all issues identified by the Python linter were fully addressed before release.

F.4 Unit Testing

The software development process for the implementation of the SRUP protocol made extensive use of unit testing to ensure the correctness of the classes, methods, and

```cpp
#include "SRUP.h"

SRUP_MSG::SRUP_MSG()
{
    m_version = new uint8_t[1];
    m_msgtype = new uint8_t[1];

    // We won't actually allocate any space for the other members in the ↙
    ↳ constructor...
    // We'll do that dynamically on assignment.

    m_version[0] = SRUP::SRUP_VERSION;

    m_sig_len = 0;
    m_unsigned_length = 0;
    m_token_len = 0;
    m_serial_length = 0;

    m_is_serialized = false;

    m_signature = nullptr;
    m_token = nullptr;
    m_serialized = nullptr;
    m_unsigned_message = nullptr;
    m_sequence_ID = nullptr;
    m_sender_ID = nullptr;
}
```

LISTING F.2: The SRUP base-class constructor function

```cpp
bool SRUP_MSG::token(const uint8_t* t, uint16_t len)
{
    try
    {
        if (len < 1)
            return false;
        else
        {
            delete[] m_token;

            m_token = new uint8_t[len];
            std::memcpy(m_token, t, len);
            m_token_len = len;
        }
    }
    catch (...)
    {
        m_token = nullptr;
        return false;
    }

    return true;
}
```

LISTING F.3: The SRUP base-class token property setter function

```
SRUP_MSG::~SRUP_MSG()
{
    delete[] m_version;
    delete[] m_msgtype;
    delete[] m_unsigned_message;
    delete[] m_signature;
    delete[] m_token;
    delete[] m_serialized;
    delete (m_sequence_ID);
    delete (m_sender_ID);
}
```

LISTING F.4: The SRUP base-class destructor function

functions providing the functionality of the library code. Specifically, the C++ classes within `libSRUP_Lib.so` were tested using the *Google Test* framework [343], and the Python classes created within `pySRUPLib.so` (although also written in C++) were tested from within Python using the *pytest* framework [344].

In each case, the tests were automatically run as a part of the build process every time the corresponding source code was changed. In addition to ensuring that the code was correct to begin with, this approach also ensured that it remained so and did not become inadvertently broken by other changes elsewhere in the codebase. Any failing tests were reported, investigated, and solved. All releases of the codebase fully passed all of the test suites.

F.4.1 C++

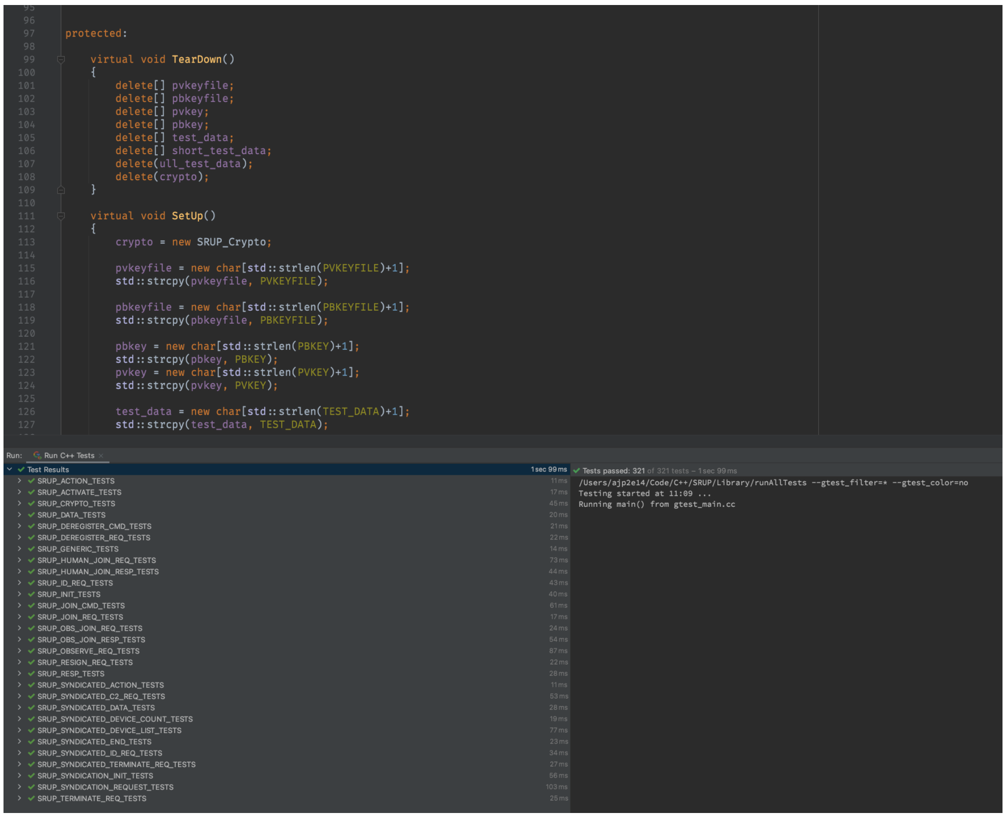
Using the Google Test framework, extensive testing was conducted to ensure the validity of the C++ code.

These tests included testing: the cryptographic methods in isolation (both the encryption and the verification of signatures); each of the message type classes' *getter* and *setter* methods; data length verification; correct serialization and deserialization; as well as testing that the class functions 'gracefully fail' when applied to invalid or incomplete data.

In keeping with good practice [345], tests written using the Google Test framework made extensive use of *test fixtures* (in the form of `SetUp()` and `TearDown()` functions), and were designed in order to test as complete a range of conditions as possible (such as missing fields, excessively large data, boundary conditions, etc.). They also tested that the generic message type (used as a placeholder when processing a received message) worked correctly when used with any full message type.

The full list of the C++ / Google Test unit tests can be seen in the file `tests.cpp` available in the `Tests` directory of the source code repository on GitHub (https://github.com/dstl/srup/blob/master/Test/tests.cpp).

A screenshot showing the C++ tests running in the CLion IDE can be seen in Figure F.1.



```
protected:

    virtual void TearDown()
    {
        delete[] pvkeyfile;
        delete[] pbkeyfile;
        delete[] pvkey;
        delete[] pbkey;
        delete[] test_data;
        delete[] short_test_data;
        delete(ull_test_data);
        delete(crypto);
    }

    virtual void SetUp()
    {
        crypto = new SRUP_Crypto;

        pvkeyfile = new char[std::strlen(PVKEYFILE)+1];
        std::strcpy(pvkeyfile, PVKEYFILE);

        pbkeyfile = new char[std::strlen(PBKEYFILE)+1];
        std::strcpy(pbkeyfile, PBKEYFILE);

        pbkey = new char[std::strlen(PBKEY)+1];
        std::strcpy(pbkey, PBKEY);
        pvkey = new char[std::strlen(PVKEY)+1];
        std::strcpy(pvkey, PVKEY);

        test_data = new char[std::strlen(TEST_DATA)+1];
        std::strcpy(test_data, TEST_DATA);
```

```
Run:  Run C++ Tests
✓ Test Results                                   1 sec 99 ms
  ✓ SRUP_ACTION_TESTS                                  11 ms
  ✓ SRUP_ACTIVATE_TESTS                                17 ms
  ✓ SRUP_CRYPTO_TESTS                                  45 ms
  ✓ SRUP_DATA_TESTS                                    20 ms
  ✓ SRUP_DEREGISTER_CMD_TESTS                          21 ms
  ✓ SRUP_DEREGISTER_REQ_TESTS                          22 ms
  ✓ SRUP_GENERIC_TESTS                                 14 ms
  ✓ SRUP_HUMAN_JOIN_REQ_TESTS                          73 ms
  ✓ SRUP_HUMAN_JOIN_RESP_TESTS                         44 ms
  ✓ SRUP_ID_REQ_TESTS                                  43 ms
  ✓ SRUP_INIT_TESTS                                    40 ms
  ✓ SRUP_JOIN_CMD_TESTS                                61 ms
  ✓ SRUP_JOIN_REQ_TESTS                                17 ms
  ✓ SRUP_OBS_JOIN_REQ_TESTS                            24 ms
  ✓ SRUP_OBS_JOIN_RESP_TESTS                           54 ms
  ✓ SRUP_OBSERVE_REQ_TESTS                             87 ms
  ✓ SRUP_RESIGN_REQ_TESTS                              22 ms
  ✓ SRUP_RESP_TESTS                                    28 ms
  ✓ SRUP_SYNDICATED_ACTION_TESTS                       11 ms
  ✓ SRUP_SYNDICATED_C2_REQ_TESTS                       53 ms
  ✓ SRUP_SYNDICATED_DATA_TESTS                         28 ms
  ✓ SRUP_SYNDICATED_DEVICE_COUNT_TESTS                 19 ms
  ✓ SRUP_SYNDICATED_DEVICE_LIST_TESTS                  77 ms
  ✓ SRUP_SYNDICATED_END_TESTS                          23 ms
  ✓ SRUP_SYNDICATED_ID_REQ_TESTS                       34 ms
  ✓ SRUP_SYNDICATED_TERMINATE_REQ_TESTS                27 ms
  ✓ SRUP_SYNDICATION_INIT_TESTS                        56 ms
  ✓ SRUP_SYNDICATION_REQUEST_TESTS                    103 ms
  ✓ SRUP_TERMINATE_REQ_TESTS                           25 ms
```

```
✓ Tests passed: 321 of 321 tests – 1 sec 99 ms
/Users/ajp2e14/Code/C++/SRUP/Library/runAllTests --gtest_filter=* --gtest_color=no
Testing started at 11:09 ...
Running main() from gtest_main.cc
```

FIGURE F.1: A screenshot showing the C++ unit test runner in CLion.

F.4.2 Python

The same approach was adopted using pytest. Each of the classes created by `pySRUPLib.so` are tested using a separate test file (adopting the standard naming convention of `<`*`classname`*`>_test.py`).

Again, the tests were designed to test *setters* and *getters*, serialization, under- and overflow, etc., as well as testing the correct processing of data messages into native Python data types.

The full list of the Python / pytest unit tests can be seen in the files contained within the directory `./pySRUP/Python/tests/` available in the source code repository on GitHub (https://github.com/dstl/srup/tree/master/pySRUP/Python/tests).

F.5 Incremental development and testing

The design and implementation approach adopted for the development of the SRUP libraries has been an incremental and iterative one. At all stages in the development work, functionality has first been prototyped in the context of software-only implementations run on a development laptop, before moving to increasingly complete implementations on representative (and then real) hardware. As described in Section F.4, all classes, methods, and functions were subjected to unit testing to ensure their operation in isolation matched the intended output. This approach has also ensured that the more complex functionality within the protocol could be evaluated (and iteratively designed) based on performance and operation.

F.6 Future Work

There are a number of additional avenues of security assessment of the SRUP protocol, and its software implementation, that could be carried out in the future in order to provide even greater assurance as to the correctness of the protocol and its robustness to attack.

F.6.1 Adversarial testing

To date, all of the assessment of the security of the protocol has been conducted by the author, as a part of this PhD research. Although an adversarial approach has been taken in the design and assessment of the protocol (see Section F.1.1), no truly independent testing has been carried out by a third-party. Whilst all efforts were made to ensure that the assessment was conducted fairly and with all due diligence, it is inevitable that individuals alone will not be able to imagine all potential threat paths. As such, true adversarial testing (also known as *red teaming*) by an independent party should be conducted in order to provide greater guarantees as to the correctness and robustness of the protocol. In addition to analysis of the protocol itself, there are a number of techniques that could be applied in order to test the software implementation — the most common approaches for an application such as this would be *fuzzing* and *reverse engineering*.

F.6.1.1 Fuzzing

Fuzzing [346] is a process whereby inputs to a piece of software are automatically generated by a test tool, in order to both establish that the software under test responds correctly to an expected range of inputs and to attempt to cause the software's input validation to fail by generating input that has unintended effects. A simple example of this may be to submit oversized and otherwise malformed input, in order to attempt to cause a buffer overflow. Depending on the structure of the software, such an overflow may cause a vulnerability permitting arbitrary code execution (see Section 2.4.1). Although this approach requires a concrete instance of the implementation of the protocol to test against, it is capable of identifying vulnerabilities in both the implementation and the underlying design of the protocol (such as the use of fixed data type sizes).

F.6.1.2 Software Reverse Engineering

Another approach targeting a specific implementation of the protocol (or other element of the software) is software reverse engineering. This (extremely time-consuming) approach takes the compiled binary form of an executable or software library, turns the binary machine code back into human-readable assembler code (*disassembly*), and then turns that assembly code into equivalent C code (*decompilation*). Once this has been done, skilled users of these tools can reconstruct the data flow within the binary code to understand its structure and composition without ever having seen the original source code. Using this inside knowledge of the construction of the code, alongside other techniques mentioned in this section, can enable the identification of implementation issues, both in the application's source code and in its interaction with any statically linked libraries.

In the context of open source software (such as the implementation of SRUP) this step can be omitted and the published source code could be analysed instead — although by analysing the reverse engineered binary it is possible to be certain that the source code wasn't changed from the published version before compilation.

A thorough code review process can then be conducted to closely examine all potential sources of errors and vulnerabilities (such as the use of input buffers, type definitions, dynamic heap variable allocation, etc.), with findings being tested either using more directed fuzzing or via hand-crafting specific adversarial inputs.

F.6.2 Formal Risk Analysis

A more rigorous approach to establishing the validity of a protocol can be taken by performing formal risk analysis. This is a process that takes formal consideration of elements of the expected use case, the constructional elements, and the data flows, and considers the impact that compromise to any part of the system may have on other elements or the system as a whole [347]. Microsoft use the mnemonic STRIDE (Spoofing, Tampering, Repudiation, Information disclosure, Denial-of-service, Elevation of privilege) to categorize the types of threats considered by their threat analysis process [348].

Such a process, whilst largely a formalization of the approach used by the adversarial design concept described in Section F.1.1, provides greater assurance that nothing has been overlooked by following a described process and by producing documentary evidence that the process has been followed. This consistent approach is also reproducible. It can also describe precisely how the system is robust to threats against each component.

F.6.3 Formal Verification

For the highest degree of assurance, formal verification of the protocol and its software implementation could be performed.

This is the most time-consuming of the approaches as it requires the formalization of both the design and the implementation, but it does offer the most comprehensive guarantees as to the correctness of the system. Using techniques such as *Z Notation* [349], one can precisely specify the requirements of the system using a mathematical notation, and using these formally prove that only permitted operations can occur. Using this formal specification, software implementations can then be developed, and can be demonstrated to have the same properties of correctness. Due to the complexities, time, and cost required to perform this type of formal verification on anything other than a trivial system, in practice these techniques are reserved for safety-critical software applications (such as medical devices [350] or flight-control systems [351]).

Acronyms

2G Second-Generation. 162

3G Third-Generation. 8, 162

4G Fourth-Generation. 8, 162, 166

5G Fifth-Generation. 8

6LoWPAN IPv6 over Low power Wireless Personal Area Networks. 7, 33

ACL Access Control List. 59, 60, 176

AES Advanced Encryption Standard. 33, 34, 175

API Application Program Interface. 37, 46, 71, 77, 78, 80, 104, 112, 129, 144, 155, 203

ARPANET Advanced Research Project Agency Network. 15, 32

ASCII American Standard Code for Information Interchange. 26, 54, 171

ATM Asynchronous Transfer Mode. 32

AWS Amazon Web Services. 80

BIOS Basic Input/Output System. 14

BLE Bluetooth Low Energy. 81

BOM Bill of Material. 139

C2 Command and Control. 3, 4, 22, 41, 43–50, 53, 56–60, 62–67, 71, 72, 75–80, 82, 83, 85–88, 95, 96, 104–108, 110–115, 121, 125, 126, 128–130, 132, 133, 136, 139, 141–145, 147–152, 154, 155, 158–161, 165, 175–179, 181, 182, 189, 195, 203, 204, 206, 208, 224

CA Certificate Authority. 34, 35, 74, 75, 77, 78, 80, 132

CN Common Name. 35, 59, 78, 80

CNI Critical National Infrastructure. 11, 23, 24

CoAP Constrained Application Protocol. 38, 39, 71, 77

CoRE Constrained RESTful Environments. 38

CPS Cyber Physical Systems. 10, 11, 23, 24, 70, 182

CPU Central Processing Unit. 6, 159, 172, 178, 180

CSR Certificate Signing Request. 35, 78, 80

dBm decibel-milliwatts. 162

DCMS Department of Digital, Culture, Media & Sport. 20, 115, 223

DDoS Distributed Denial of Service. 17, 21, 22, 111

DNS Domain Name System. 22

DoD United States' Department of Defense. 13, 14, 32

DoE United States Department of Energy. 14

DOS Disk Operating System. 14, 19

DoS Denial of Service. 15, 17, 21, 48, 80, 111, 114

Dstl the Defence Science and Technology Laboratory. xxi, 225

DTLS Datagram Transport Layer Security. 35, 38, 102

E2EE End-to-End Encrypted. 45

EC2 Elastic Cloud Compute. 80

ECC Elliptic Curve Cryptography. 28

ECDH Elliptic Curve Diffie-Hellman. 28

ECDHE Elliptic Curve Diffie-Hellman Ephemeral. 34

ECDSA Elliptic Curve Digital Signature Algorithm. 36

EDGE Enhanced Data Rates for GSM Evolution. 162

EEPROM Electrically Erasable Programmable Read Only Memory. 78, 110, 136

eInk Electronic Ink. 94, 139

FQDN Fully Qualified Domain Name. 36

FTP File Transfer Protocol. 32

GC&CS Government Code and Cipher School. 25

GCHQ Government Communications Headquarters. 25, 28

GPRS General Packet Radio Service. 162

GSM Global System for Mobile Communications. 162

HAT Hardware Attached on Top. 136

HCI Human-Computer Interface. 9, 82

HPC High Performance Computing. 6, 32

HTTP Hyper-Text Transfer Protocol. 6, 15, 32, 34, 37–39, 69, 71, 108, 111, 124, 125, 190, 224, 225

HTTPS Secure Hyper-Text Transfer Protocol. 33, 34, 37, 46, 68, 70, 72, 75–77, 80, 82, 102, 114, 158, 197, 203, 224, 225

I^{2}C Inter-Integrated Circuit Protocol. 100, 101, 136

IDE Integrated Development Environment. 225, 229

IIoT Industrial Internet of Things. 8, 23

IIS Internet Information Services. 15

IoBT Internet of Battle Things. 9–11, 46, 47, 141

IoT Internet of Things. 3–11, 13, 20–24, 28, 29, 31, 37, 38, 41, 43, 45–50, 53, 54, 70, 72, 73, 76, 82, 84, 101, 102, 105–108, 113, 114, 116, 125, 133, 135, 139, 141, 142, 145, 151, 152, 155, 158, 161, 166, 168, 170, 172, 175, 178–182, 225

IP Internet Protocol. 7, 8, 32, 33, 38, 186

IPSec Internet Protocol Security. 33

IPv4 Internet Protocol Version 4. 31

IPv6 Internet Protocol Version 6. 7, 8, 31, 33, 39

ITU International Telecommunication Union. 34

JSON JavaScript Object Notation. 54, 70–72, 78, 80

kb/s kilobits per second. 162

LAN Local Area Network. 162, 164, 166

LCD Liquid Crystal Display. 90, 92, 97, 136, 139

LED Light Emitting Diode. 125, 133, 158

LPDDR2 second generation Low-Power Double Data Rate. 159

LPDDR4 fourth generation Low-Power Double Data Rate. 159

M2M Machine-to-Machine. 7–9, 37

MACA Military Aid to the Civil Authorities. 9

MaDIoT Manipulation of Demand via Internet of Things. 23

MQTT Message Queuing Telemetry Transport. 6, 31, 32, 38–41, 53, 54, 56–60, 68–72, 76–78, 80, 82, 86, 87, 102, 111, 115, 116, 119, 124–126, 129, 130, 132, 133, 139, 146, 147, 157–159, 164–166, 168–171, 173, 175, 176, 178, 180, 183–187, 189, 195–198, 201–203, 225

NASA National Aeronautics and Space Administration. 6

NAT Network Address Translation. 8, 39

NATO North Atlantic Treaty Organization. 43

NCSC National Cyber Security Centre. 20, 115

NDEF NFC Data Exchange Format. 100

NFC Near-Field Communication. 81, 100, 101, 103–105, 139, 176

NGO Non-Governmental Organization. 17, 141, 144

NHS UK National Health Service. 16

NIST the United States' National Institute of Standards and Technology. 33

NSA United States National Security Agency. 14

NTP Network Time Protocol. 108, 160

OLED Organic Light Emitting Diode. 90

OS Operating System. 14, 15, 18, 179, 180

OSS Open Source Software. 6, 111, 119, 129, 155, 182

PARC Palo Alto Research Center. 7, 32

PC Personal Computer. 14

PCB Printed Circuit Board. 136

PGP Pretty Good Privacy. 95

PKCS#1 Public-Key Cryptography Standard #1. 57

PROM Programmable Read Only Memory. 74

PUF Physical Unclonable Function. 74

QoS Quality of Service. 38, 40, 69, 185, 187

QUIC Quick UDP Internet Connections. 35

RAM Random Access Memory. 159

RAT Remote Access Trojan. 17

REST Representational State Transfer. 37, 38, 46, 77, 78, 80, 112, 129, 130, 155

RF Radio Frequency. 17, 18, 97, 105, 180

RFID Radio-Frequency Identification. 7, 100, 139

ROM Read Only Memory. 14, 74, 136

RSA Rivest-Shamir-Adleman. 27, 28, 33, 34, 36, 53, 54, 84, 111, 113, 119, 157, 158, 171, 175, 197, 224, 225

RSRP Reference Signals Received Power. 162

SDRAM Synchronous Dynamic Random-Access Memory. 159

SFTP SSH File Transfer Protocol. 37

SHA Secure Hash Algorithm. 111

SHA-2 Secure Hash Algorithm, version 2. 33, 34, 53

SHA-256 256-byte SHA-2. 33, 36, 54, 69, 109–111, 114, 115, 157, 175

SNEP Simple NDEF Exchange Protocol. 100

SPI Serial Peripheral Interface. 136

SQL Structured Query Language. 18, 130

SRUP Secure Remote Update Protocol. 53–73, 76–78, 83, 84, 87, 92, 96, 97, 100, 102, 105–107, 109, 110, 112–116, 119–126, 129–131, 133–135, 139, 141, 143–146, 148, 150–152, 154, 155, 157–159, 162–164, 166, 168–173, 175–182, 189–195, 197, 200, 202–204, 206, 208, 215, 223–226, 229–231

SSH Secure Shell. 32, 36, 37, 95

SSL Secure Sockets Layer. 27, 34

STRIDE Spoofing, Tampering, Repudiation, Information disclosure, Denial-of-service, Elevation of privilege. 231

TCP Transmission Control Protocol. 31, 32, 35, 39, 40, 58, 108, 111, 175, 185, 186

TCP/IP Internet Protocol Suite. 31, 39

TLS Transport Layer Security. 6, 27, 33–35, 41, 53, 59, 77, 78, 80, 111, 115, 130, 133, 158, 159, 171, 175, 183, 203

TLS-PSK Transport Layer Security using a Pre-Shared Key. 41

TPM Trusted Platform Module. 74, 116

UDP User Datagram Protocol. 31, 32, 35, 38, 39

UML Unified Modeling Language. 121

URI Uniform Resource Identifier. 38, 75

URL Universal Resource Locator. 35, 63, 71, 76–78, 81, 82, 102, 125, 129, 147, 176, 190, 203

US ARL United States Army Research Laboratory. 9, 46

USAF United States Air Force. 14, 18

USB Universal Serial Bus. 101, 103, 164, 170

UUID Universally Unique Identifier. 73, 74, 78, 90, 97, 109

VoIP Voice over Internet Protocol. 33

VPN Virtual Private Network. 33

WSGI Web-Server Gateway Interface. 131

X.509 International Telecommunication Union X.509. 34, 35, 59, 77, 225

XOR Exclusive OR. 26

YAML *YAML Ain't Markup Language*. 130

References

[1] A. J. Poulter, S. J. Johnston, and S. J. Cox, "Using the MEAN Stack to Implement a RESTful Service for an Internet of Things Application," in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, Jan. 2015, pp. 280–285. DOI: 10.1109/WF-IoT.2015.7389066.

[2] ——, "SRUP: The Secure Remote Update Protocol," in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, IEEE, Jan. 2016, pp. 42–47. DOI: 10.1109/WF-IoT.2016.7845397.

[3] ——, "Extensions and Enhancements to The Secure Remote Update Protocol," *Future Internet*, vol. 9, no. 4, p. 59, Sep. 2017. DOI: 10.3390/fi9040059.

[4] ——, "pySRUP – Simplifying Secure Communications for Command Control in the Internet of Things," in *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*, Apr. 2019, pp. 273–277. DOI: 10.1109/WF-IoT.2019.8767205.

[5] A. J. Poulter, S. Johnston, and S. Cox, "Secure Messaging, Key Management & Device identity for the IoT," in *Presentation to IoT Security Foundation Conference 2019*, IoT Security Foundation, Nov. 2019. [Online]. Available: https://youtu.be/vdjY617WvHo.

[6] A. J. Poulter, S. J. Ossont, and S. J. Cox, "Enabling the Secure Use of Dynamic Identity for the Internet of Things—Using the Secure Remote Update Protocol (SRUP)," *Future Internet*, vol. 12, no. 8, p. 138, Aug. 2020. DOI: 10.3390/fi12080138.

[7] A. J. Poulter and S. J. Cox, "Enabling secure guest access for Command-and-Control of Internet of Things devices," *IoT*, vol. 2, no. 2, pp. 236–248, 2021. DOI: 10.3390/iot2020013.

[8] ——, "An assessment of the performance of the secure remote update protocol in simulated real-world conditions," *IoT*, vol. 2, no. 4, pp. 549–563, 2021. DOI: 10.3390/iot2040028.

[9] K. L. Lueth, *State of the IoT 2020: 12 billion IoT connections, surpassing non-IoT for the first time*, Nov. 2020. [Online]. Available: https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non-iot-for-the-first-time/.

[10] T. Sterling, D. Savarese, D. J. Becker, B. Fryxell, and K. Olson, "Communication Overhead for Space Science Applications on the Beowulf Parallel Workstation," in *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing*, Aug. 1995, pp. 23–30. DOI: 10.1109/HPDC.1995.518691.

[11] S. J. Cox, D. A. Nicole, and K. Takeda, "Commodity High Performance Computing at Commodity Prices," in *Proceedings of the 21st World Occam and Transputer User Group Technical Meeting*, IOS Press, 1998.

[12] J. Dongarra, E. Strohmaier, H. Simon, and M. Meuer, *November 2020: TOP500 List*, https://top500.org/lists/top500/2020/11/, Nov. 2020. [Online]. Available: https://top500.org/lists/top500/2020/11.

[13] G. Halfacree and E. Upton, *Raspberry Pi User Guide*, 1st ed. Wiley Publishing, 2012, ISBN: 111846446X.

[14] B. Balon and M. Simić, "Using Raspberry Pi Computers in Education," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2019, pp. 671–676. DOI: 10.23919/MIPRO.2019.8756967.

[15] V. Vujović and M. Maksimović, "Raspberry Pi as a Sensor Web node for home automation," *Computers and Electrical Engineering*, vol. 44, pp. 153–171, 2015. DOI: https://doi.org/10.1016/j.compeleceng.2015.01.019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0045790615000257.

[16] T. Sorwar, S. B. Azad, S. R. Hussain, and A. I. Mahmood, "Real-time Vehicle monitoring for traffic surveillance and adaptive change detection using Raspberry Pi camera module," in *2017 IEEE Region 10 Humanitarian Technology Conference (R10-HTC)*, Dec. 2017, pp. 481–484. DOI: 10.1109/R10-HTC.2017.8289003.

[17] A. K. Saha, S. Roy, A. Bhattacharya, P. Shankar, A. K. Sarkar, H. N. Saha, and P. Dasgupta, "A low cost remote controlled underwater rover using Raspberry Pi," in *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, Jan. 2018, pp. 769–772. DOI: 10.1109/CCWC.2018.8301657.

[18] S. J. Cox, J. T. Cox, R. P. Boardman, S. J. Johnston, M. Scott, and N. S. O'Brien, "Iridis-pi: A low-cost, compact demonstration cluster," *Cluster Computing*, vol. 17, pp. 349–358, Jun. 2013. DOI: https://doi.org/10.1007/s10586-013-0282-7.

[19] D. Cassel, *The Hardware and Software Used in Space*, https://thenewstack.io/the-hardware-and-software-used-in-space/, Oct. 2020. [Online]. Available: https://thenewstack.io/the-hardware-and-software-used-in-space/.

[20] J. Delaune, R. Brockers, D. S. Bayard, H. Dor, R. Hewitt, J. Sawoniewicz, G. Kubiak, T. Tzanetos, L. Matthies, and J. Balaram, "Extended Navigation Capabilities for a Future Mars Science Helicopter Concept," in *2020 IEEE Aerospace Conference*, 2020, pp. 1–10. DOI: 10.1109/AERO47225.2020.9172289.

[21] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “Hypertext Transfer Protocol – HTTP/1.1,” Internet Engineering Task Force, Tech. Rep. RFC2616, Jun. 1, 1999. [Online]. Available: https://tools.ietf.org/html/rfc2616.

[22] OpenSSL Software Foundation, *OpenSSL - Cryptography and SSL/TLS Toolkit*. [Online]. Available: https://www.openssl.org/ (visited on 04/06/2018).

[23] E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.3,” Internet Engineering Task Force, Standard RFC 8446, Aug. 2018. [Online]. Available: https://tools.ietf.org/html/rfc8446 (visited on 03/15/2021).

[24] A. Banks and R. Gupta, “MQTT Version 3.1.1,” *Oasis Standard*, A. Banks and R. Gupta, Eds., Oct. 29, 2014. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

[25] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, “Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications,” *IEEE Communications Surveys Tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015. DOI: 10.1109/COMST.2015.2444095.

[26] W. A. Jabbar, H. K. Shang, S. N. I. S. Hamid, A. A. Almohammedi, R. M. Ramli, and M. A. H. Ali, “IoT-BBMS: Internet of Things-Based Baby Monitoring System for Smart Cradle,” *IEEE Access*, vol. 7, pp. 93 791–93 805, 2019. DOI: 10.1109/ACCESS.2019.2928481.

[27] M. Weiser, “The Computer for the 21st-Century,” *Scientific American*, vol. 265, no. 3, pp. 94–104, Sep. 1991. DOI: 10.1038/scientificamerican0991-94.

[28] R. Want, B. N. Schilit, N. I. Adams, R. Gold, K. Petersen, D. Goldberg, J. R. Ellis, and M. Weiser, “An Overview of the PARCTAB Ubiquitous Computing Experiment,” *IEEE Personal Communications*, vol. 2, no. 6, pp. 28–43, Dec. 1995. DOI: 10.1109/98.475986.

[29] G. Montenegro, N. Kushalnagar, J. W. Hui, and D. E. Culler, *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*, IETF, Sep. 2007. [Online]. Available: https://tools.ietf.org/html/rfc4944.

[30] S. Deering and R. Hinden, *Internet Protocol, Version 6 (IPv6) Specification*, Internet Engineering Task Force, Jul. 2017. [Online]. Available: https://tools.ietf.org/html/rfc8200.

[31] IEEE Computer Society, *IEEE Standard for Local and metropolitan area networks—Part 15.4: Low-Rate Wireless Personal Area Networks (WPANs)*, Sep. 2011. [Online]. Available: https://standards.ieee.org/getieee802/download/802.15.4-2011.pdf.

[32] C. Everhart, E. Caplan, and R. Frederking, *The "Only" Coke Machine on the Internet*, Jun. 1998. [Online]. Available: https://www.cs.cmu.edu/~coke/history_long.txt (visited on 03/22/2021).

[33] E. Perahia, “IEEE 802.11n Development: History, Process, and Technology,” *IEEE Communications Magazine*, vol. 46, no. 7, pp. 48–55, 2008. DOI: 10.1109/MCOM.2008.4557042.

[34] C. R. Schoenberger, “The internet of things,” *Forbes*, Mar. 2002. [Online]. Available: http://www.forbes.com/global/2002/0318/092.html.

[35] K. Ashton. (Jun. 2009). “That ‘Internet of Things’ Thing - RFID Journal,” [Online]. Available: http://www.rfidjournal.com/articles/view?4986.

[36] L. Atzori, A. Iera, and G. Morabito, “The Internet of Things: A survey,” *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010. DOI: 10.1016/j.comnet.2010.05.010.

[37] G. Lawton, “Machine-to-machine technology gears up for growth,” *Computer*, vol. 37, no. 9, pp. 12–15, 2004. DOI: 10.1109/MC.2004.137.

[38] P. Lade, R. Ghosh, and S. Srinivasan, “Manufacturing Analytics and Industrial Internet of Things,” *IEEE Intelligent Systems*, vol. 32, no. 3, pp. 74–79, May 2017. DOI: 10.1109/MIS.2017.49. [Online]. Available: http://ieeexplore.ieee.org/document/7933925/.

[39] B. Schneier, *Click Here to Kill Everyone*. New York, New York, USA: W. W. Norton & Company, 2018, ISBN: 0-393-60888-3.

[40] S. Sorrell, “’Internet of Things’ connected deviuces to triple by 2021, reaching over 46 Billion units,” Juniper Research, Whitepaper, Dec. 2016. [Online]. Available: https://www.juniperresearch.com/press/press-releases/%E2%80%98internet-of-things%E2%80%99-connected-devices-triple-2021.

[41] M. Rothmuller and S. Barker, “IoT   The Internet of Transformation 2020,” Juniper Research, White paper, Apr. 2020.

[42] D. Maimon, M. Becker, S. Patil, and J. Katz, “Self-protective behaviors over public WiFi networks,” in *The LASER workshop: Learning from authoritative security experiment results (LASER 2017)*, 2017, pp. 69–76.

[43] J. Foust, “SpaceX’s space-Internet woes: Despite technical glitches, the company plans to launch the first of nearly 12,000 satellites in 2019,” *IEEE Spectrum*, vol. 56, no. 1, pp. 50–51, 2019. DOI: 10.1109/MSPEC.2019.8594798.

[44] T. Wei, W. Feng, Y. Chen, C.-X. Wang, N. Ge, and J. Lu, “Hybrid Satellite-Terrestrial Communication Networks for the Maritime Internet of Things: Key Technologies, Opportunities, and Challenges,” *IEEE Internet of Things Journal*, pp. 1–1, 2021. DOI: 10.1109/JIOT.2021.3056091.

[45] T. Duan and V. Dinavahi, “Starlink Space Network Enhanced Cyber-Physical Power System,” *IEEE Transactions on Smart Grid*, pp. 1–1, 2021. DOI: 10.1109/TSG.2021.3068046.

[46] J. H. Chang, M. Tabassum, U. Qidwai, S. B. A. Kashem, P. Suresh, and U. Saravanakumar, “Design and Evaluate Low-Cost Wireless Sensor Network Infrastructure to Monitor the Jetty Docking Area in Rural Areas,” in *Advances in Smart System Technologies*, Springer, 2021, pp. 689–700.

[47] D. Wing, “Network Address Translation: Extending the Internet Address Space,” *IEEE Internet Computing*, vol. 14, no. 4, pp. 66–70, Jun. 2010. DOI: 10.1109/MIC.2010.96. [Online]. Available: http://ieeexplore.ieee.org/document/5496805/.

[48] S. Sreeraj and G. S. Kumar, “Performance of IoT protocols under constrained network, a Use Case based approach,” in *2018 International Conference on Communication, Computing and Internet of Things (IC3IoT)*, 2018, pp. 495–498. DOI: 10.1109/IC3IoT.2018.8668105.

[49] A. Pandharipande, M. Zhao, E. Frimout, and P. Thijssen, “IoT lighting: Towards a connected building eco-system,” in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, 2018, pp. 664–669. DOI: 10.1109/WF-IoT.2018.8355196.

[50] A. Kott, A. Swami, and B. J. West, “The Internet of Battle Things,” *IEEE Computer*, vol. 49, no. 12, pp. 70–75, Dec. 2016. DOI: 10.1109/MC.2016.355.

[51] A. K. Cebrowski and J. H. Garstka, “Network-Centric Warfare - Its Origin and Future,” *Proceedings of the United States Naval Institute*, Jan. 1998. [Online]. Available: https://www.usni.org/magazines/proceedings/1998-01/network-centric-warfare-its-origin-and-future.

[52] M. J. Farooq and Q. Zhu, “On the Secure and Reconfigurable Multi-Layer Network Design for Critical Information Dissemination in the Internet of Battlefield Things (IoBT),” *IEEE Transactions on Wireless Communications*, vol. 17, no. 4, pp. 2618–2632, 2018. DOI: 10.1109/TWC.2018.2799860.

[53] G. J. Gagnon, C. W. McLeod, and D. Thompson, “Space as a War-fighting Domain,” *Air & Space Power Journal*, vol. 32, no. 2, pp. 4–9, 2018.

[54] P. M. Nakasone and C. Lewis, “Cyberspace in multi-domain battle,” *The Cyber Defense Review*, vol. 2, no. 1, pp. 15–26, 2017.

[55] D. E. Zheng and W. A. Carter, “Leveraging the Internet of Things for a more Efficient and Effective Military,” Center for Stategic & International Studies, Tech. Rep., Sep. 2015. [Online]. Available: https://csis-prod.s3.amazonaws.com/s3fs-public/legacy_files/files/publication/150915_Zheng_LeveragingInternet_WEB.pdf.

[56] UK Ministry of Defence, *Military Aid to the Civil Authorities for activities in the UK*, Policy Paper, Aug. 2016. [Online]. Available: https://www.gov.uk/government/publications/2015-to-2020-government-policy-military-aid-to-the-civil-authorities-for-activities-in-the-uk.

[57] W. Wolf, “Cyber-Pysical Systems,” *Computer*, vol. 42, no. 3, pp. 88–89, 2009. DOI: 10.1109/MC.2009.81.

[58] E. K. Wang, Y. Ye, X. Xu, S. M. Yiu, L. C. K. Hui, and K. P. Chow, "Security Issues and Challenges for Cyber Physical System," in *Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, IEEE, Mar. 2011, pp. 733–738, ISBN: 978-1-4244-9779-9. DOI: 10.1109/GreenCom-CPSCom.2010.36. [Online]. Available: http://ieeexplore.ieee.org/document/5724910.

[59] S. K. Khaitan and J. D. McCalley, "Design Techniques and Applications of Cyberphysical Systems: A Survey," *IEEE Systems Journal*, vol. 9, no. 2, pp. 350–365, 2015. DOI: 10.1109/JSYST.2014.2322503.

[60] M. Walport, "The Internet of Things," Tech. Rep. GS/14/1230, Dec. 2014.

[61] Civil Contingencies Secretariat, "Strategic framework and policy statement on improving the resilience of critical infrastructure to disruption from natural hazards," Cabinet Office, London, Tech. Rep., Mar. 2010. [Online]. Available: https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/62504/strategic-framework.pdf.

[62] M. Sparkes, "How do we solve the problem of ransomware?" *New Scientist*, vol. 250, no. 3336, p. 13, 2021, ISSN: 0262-4079. DOI: https://doi.org/10.1016/S0262-4079(21)00899-X.

[63] R. A. Kemmerer, "Cybersecurity," in *25th International Conference on Software Engineering, 2003. Proceedings.*, 2003, pp. 705–715. DOI: 10.1109/ICSE.2003.1201257.

[64] R. von Solms and J. van Niekerk, "From information security to cyber security," *Computers & Security*, vol. 38, pp. 97–102, 2013, Cybercrime in the Digital Economy, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2013.04.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404813000801.

[65] M. Warner, "Cybersecurity: A Pre-history," *Intelligence and National Security*, vol. 27, no. 5, pp. 781–799, 2012. DOI: 10.1080/02684527.2012.708530. eprint: https://doi.org/10.1080/02684527.2012.708530. [Online]. Available: https://doi.org/10.1080/02684527.2012.708530.

[66] K. A. Minihan, "Statement to the Senate Governmental Affairs Committee Hearing on Vulnerabilities of the National Information Infrastructure," Jun. 1998. [Online]. Available: https://www.hsgac.senate.gov/imo/media/doc/minihan.pdf.

[67] C. Brown, D. Lee, C. Scott, and D. Strunk, *American cyber insecurity: The growing danger of cyber attacks*, 2014.

[68] J. D. Boys, "The Clinton administration's development and implementation of cybersecurity strategy (1993–2001)," *Intelligence and National Security*, vol. 33, no. 5, pp. 755–770, 2018. DOI: 10.1080/02684527.2018.1449369. eprint: https://doi.org/10.1080/02684527.2018.1449369. [Online]. Available: https://doi.org/10.1080/02684527.2018.1449369.

[69] N. Milosevic, “History of Malware,” *Digital forensics*, vol. 1, no. 16, pp. 58–66, Aug. 2013.

[70] A. Abusitta, M. Q. Li, and B. C. M. Fung, “Malware classification and composition analysis: A survey of recent developments,” *Journal of Information Security and Applications*, vol. 59, p. 102 828, 2021, ISSN: 2214-2126. DOI: 10.1016/j.jisa.2021.102828.

[71] G. H. Khanaka and W. J. Orvis, “Virus Information Update,” Department of Engergy – Computer Incident Advisory Capability, Oak Ridge, Tennessee, USA, Tech. Rep. CIAC-2301, May 1998. eprint: https://apps.dtic.mil/sti/pdfs/ADA394231.pdf.

[72] R. van Heerden, H. Pieterse, and B. Irwin, “Mapping the Most Significant Computer Hacking Events to a Temporal Computer Attack Model,” in *ICT Critical Infrastructures and Society*, M. D. Hercheui, D. Whitehouse, W. McIver, and J. Phahlamohlaka, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 226–236, ISBN: 978-3-642-33332-3.

[73] V. Bontchev, “Possible macro virus attacks and how to prevent them,” *Computers & Security*, vol. 15, no. 7, pp. 595–626, 1996, ISSN: 0167-4048. DOI: https://doi.org/10.1016/S0167-4048(97)88131-X. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016740489788131X.

[74] P. Docherty and P. Simpson, “Macro attacks: What next after Melissa?” *Computers & Security*, vol. 18, no. 5, pp. 391–395, 1999, ISSN: 0167-4048. DOI: https://doi.org/10.1016/S0167-4048(99)80084-4. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404899800844.

[75] A. Afianian, S. Niksefat, B. Sadeghiyan, and D. Baptiste, “Malware Dynamic Analysis Evasion Techniques: A Survey,” *ACM Comput. Surv.*, vol. 52, no. 6, Nov. 2019, ISSN: 0360-0300. DOI: 10.1145/3365001. [Online]. Available: https://doi.org/10.1145/3365001.

[76] H. Orman, “The Morris worm: a fifteen-year perspective,” *IEEE Security Privacy*, vol. 1, no. 5, pp. 35–43, 2003. DOI: 10.1109/MSECP.2003.1236233.

[77] T. Eisenberg, D. Gries, J. Hartmanis, D. Holcomb, M. S. Lynn, and T. Santoro, “The Cornell Commission: On Morris and the Worm,” *Commun. ACM*, vol. 32, no. 6, pp. 706–709, Jun. 1989, ISSN: 0001-0782. DOI: 10.1145/63526.63530. [Online]. Available: https://doi.org/10.1145/63526.63530.

[78] S. Furnell and E. H. Spafford, “The Morris Worm at 30,” *ITNOW*, vol. 61, no. 1, pp. 32–33, Feb. 2019, ISSN: 1746-5702. DOI: 10.1093/itnow/bwz013. eprint: https://academic.oup.com/itnow/article-pdf/61/1/32/28269379/bwz013.pdf. [Online]. Available: https://doi.org/10.1093/itnow/bwz013.

[79] H. Berghel, “The Code Red worm,” *Communications of the ACM*, vol. 44, no. 12, pp. 15–19, 2001.

[80] D. Moore, C. Shannon, and K. Claffy, “Code-Red: A Case Study on the Spread and Victims of an Internet Worm,” in *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment*, ser. IMW ’02, Marseille, France: Association for Computing Machinery, 2002, pp. 273–284, ISBN: 158113603X. DOI: 10.1145/637201.637244. [Online]. Available: https://doi.org/10.1145/637201.637244.

[81] J. S. Aidan, H. K. Verma, and L. K. Awasthi, “Comprehensive Survey on Petya Ransomware Attack,” in *2017 International Conference on Next Generation Computing and Information Systems (ICNGCIS)*, 2017, pp. 122–125. DOI: 10.1109/ICNGCIS.2017.30.

[82] A. Greenberg, “The untold story of NotPetya, the most devastating cyberattack in history,” *Wired, August*, vol. 22, 2018.

[83] S. Y. A. Fayi, “What Petya/NotPetya Ransomware Is and What Its Remidiations Are,” in *Information Technology - New Generations*, S. Latifi, Ed., Cham: Springer International Publishing, 2018, pp. 93–100, ISBN: 978-3-319-77028-4.

[84] S. B. Wicker, “The Ethics of Zero-Day Exploits—: The NSA Meets the Trolley Car,” *Commun. ACM*, vol. 64, no. 1, pp. 97–103, Dec. 2020, ISSN: 0001-0782. DOI: 10.1145/3393670. [Online]. Available: https://doi.org/10.1145/3393670.

[85] M. R. Faghani, A. Matrawy, and C.-H. Lung, “A Study of Trojan Propagation in Online Social Networks,” in *2012 5th International Conference on New Technologies, Mobility and Security (NTMS)*, 2012, pp. 1–5. DOI: 10.1109/NTMS.2012.6208767.

[86] S. Mansfield-Devine, “When advertising turns nasty,” *Network Security*, vol. 2015, no. 11, pp. 5–8, 2015, ISSN: 1353-4858. DOI: https://doi.org/10.1016/S1353-4858(15)30098-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1353485815300982.

[87] M. Hypponen, “Malware goes mobile,” *Scientific American*, vol. 295, no. 5, pp. 70–77, 2006. [Online]. Available: http://www.jstor.org/stable/26069041.

[88] C. G. J. Putman, S. Abhishta, and L. J. M. Nieuwenhuis, “Business Model of a Botnet,” in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2018, pp. 441–445. DOI: 10.1109/PDP2018.2018.00077.

[89] S. Greengard, “The Worsening State of Ransomware,” *Communications of the ACM*, vol. 64, no. 4, pp. 15–17, Mar. 2021, ISSN: 0001-0782. DOI: 10.1145/3449054. [Online]. Available: https://doi.org/10.1145/3449054.

[90] A. L. Young and M. Yung, “On Ransomware and Envisioning the Enemy of Tomorrow,” *IEEE Computer*, vol. 50, no. 11, pp. 82–85, 2017. DOI: 10.1109/MC.2017.4041366.

[91] 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)*, 2017, pp. 454–460. DOI: 10.1109/ICMLA.2017.0-119.

[92] M. Burgess, "Everything you need to know about EternalBlue–the NSA exploit linked to Petya," *WIRED, WIRED UK*, vol. 29, 2017.

[93] H. Orman, "Evil Offspring - Ransomware and Crypto Technology," *IEEE Internet Computing*, vol. 20, no. 5, pp. 89–94, 2016. DOI: 10.1109/MIC.2016.90.

[94] S. S. C. Silva, R. M. P. Silva, R. C. G. Pinto, and R. M. Salles, "Botnets: A survey," *Computer Networks*, vol. 57, no. 2, pp. 378–403, 2013, Botnet Activity: Analysis, Detection and Shutdown, ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2012.07.021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128612003568.

[95] P. Wainwright and H. Kettani, "An Analysis of Botnet Models," in *Proceedings of the 2019 3rd International Conference on Compute and Data Analysis*, ser. ICCDA 2019, Kahului, HI, USA: Association for Computing Machinery, 2019, pp. 116–121, ISBN: 9781450366342. DOI: 10.1145/3314545.3314562. [Online]. Available: https://doi.org/10.1145/3314545.3314562.

[96] A. Zimba, Z. Wang, M. Mulenga, and N. H. Odongo, "Crypto Mining Attacks in Information Systems: An Emerging Threat to Cyber Security," *Journal of Computer Information Systems*, vol. 60, no. 4, pp. 297–308, 2020. DOI: 10.1080/08874417.2018.1477076. eprint: https://doi.org/10.1080/08874417.2018.1477076. [Online]. Available: https://doi.org/10.1080/08874417.2018.1477076.

[97] E. Bertino and N. Islam, "Botnets and Internet of Things Security," *Computer*, vol. 50, no. 2, pp. 76–79, 2017. DOI: 10.1109/mc.2017.62. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=7842850&contentType=Journals+%26+Magazines.

[98] İ. Kara and M. Aydos, "The Ghost In The System: Technical analysis of Remote Acccess Trojan," *International Journal on Information Technologies & Security*, vol. 11, no. 1, pp. 73–84, 2019.

[99] M. Libicki, "The coming of cyber espionage norms," in *2017 9th International Conference on Cyber Conflict (CyCon)*, 2017, pp. 1–17. DOI: 10.23919/CYCON.2017.8240325.

[100] F. Ullah, M. Edwards, R. Ramdhany, R. Chitchyan, M. A. Babar, and A. Rashid, "Data exfiltration: A review of external attack vectors and countermeasures," *Journal of Network and Computer Applications*, vol. 101, pp. 18–54, 2018, ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2017.10.016.

[101] J. Menn, *Cult of the Dead Cow: How the Original Hacking Supergroup Might Just Save the World*, 1st ed. Hachette Book Group, 2019, ISBN: 154176238X.

[102] M. Rezaeirad, B. Farinholt, H. Dharmdasani, P. Pearce, K. Levchenko, and D. McCoy, "Schrödinger's RAT: Profiling the stakeholders in the remote access trojan ecosystem," in *27th USENIX security symposium (USENIX Security 18)*, 2018, pp. 1043–1060.

[103] S. Le Blond, A. Uritesc, C. Gilbert, Z. L. Chua, P. Saxena, and E. Kirda, "A look at targeted attacks through the lens of an NGO," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 543–558.

[104] T. Mahjabin, Y. Xiao, G. Sun, and W. Jiang, "A survey of distributed denial-of-service attack, prevention, and mitigation techniques," *International Journal of Distributed Sensor Networks*, vol. 13, no. 12, p. 1 550 147 717 741 463, 2017. DOI: 10.1177/1550147717741463. eprint: https://doi.org/10.1177/1550147717741463. [Online]. Available: https://doi.org/10.1177/1550147717741463.

[105] L. Garber, "Denial-of-service attacks rip the internet," *Computer*, vol. 33, no. 4, pp. 12–17, 2000. DOI: 10.1109/MC.2000.839316.

[106] 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. DOI: 10.1109/MC.2017.201.

[107] D. Czerwinski, J. Nowak, and S. Przylucki, "Evaluation of the Jammers Performance in the WiFi Band," in *Computer Networks*, P. Gaj, M. Sawicki, G. Suchacka, and A. Kwiecień, Eds., Cham: Springer International Publishing, 2018, pp. 171–182, ISBN: 978-3-319-92459-5.

[108] W. Yu, X. Liang, Y. Sun, J. Luo, and S. Xin, "Study of Fire Control Radar Technology Countering Electronic Attack," in *Man-Machine-Environment System Engineering*, S. Long and B. S. Dhillon, Eds., Singapore: Springer Singapore, 2020, pp. 531–538.

[109] M. D. Hill, J. Masters, P. Ranganathan, P. Turner, and J. L. Hennessy, "On the Spectre and Meltdown Processor Security Vulnerabilities," *IEEE Micro*, vol. 39, no. 2, pp. 9–19, 2019. DOI: 10.1109/MM.2019.2897677.

[110] K. Tsipenyuk, B. Chess, and G. McGraw, "Seven pernicious kingdoms: A taxonomy of software security errors," *IEEE Security Privacy*, vol. 3, no. 6, pp. 81–84, 2005. DOI: 10.1109/MSP.2005.159.

[111] K. Chen, S. Zhang, Z. Li, Y. Zhang, Q. Deng, S. Ray, and Y. Jin, "Internet-of-Things Security and Vulnerabilities: Taxonomy, Challenges, and Practice," *Journal of Hardware and Systems Security*, vol. 2, no. 2, pp. 97–110, 2018, ISSN: 2509-3436. DOI: 10.1007/s41635-017-0029-7.

[112] Ș. Nicula and R. D. Zota, “Exploiting stack-based buffer overflow using modern day techniques,” *Procedia Computer Science*, vol. 160, pp. 9–14, 2019, The 10th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2019) / The 9th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2019) / Affiliated Workshops, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2019.09.437.

[113] G. Mullen and L. Meany, “Assessment of Buffer Overflow Based Attacks On an IoT Operating System,” in *2019 Global IoT Summit (GIoTS)*, 2019, pp. 1–6. DOI: 10.1109/GIOTS.2019.8766434.

[114] Y. Roumani and J. Nwankpa, “Examining Exploitability Risk of Vulnerabilities: A Hazard Model,” *Communications of the Association for Information Systems*, vol. 46, 2020, ISSN: 1529-3181. DOI: 10.17705/1CAIS.04618.

[115] J. P. Anderson, “Computer Security Technology Planning Study,” United States Air Force, Bedford Massachusetts, USA, Tech. Rep. ESD-TR-73-51, Volume 2, Oct. 1972. eprint: http://csrc.nist.gov/publications/history/ande72.pdf.

[116] E. H. Spafford, “The Internet Worm Program: An Analysis,” *SIGCOMM Comput. Commun. Rev.*, vol. 19, no. 1, pp. 17–57, Jan. 1989, ISSN: 0146-4833. DOI: 10.1145/66093.66095. [Online]. Available: https://doi.org/10.1145/66093.66095.

[117] E. Levy, “Smashing the Stack for Fun and Profit,” *Phrack*, vol. 49, no. 14, 1996. [Online]. Available: http://phrack.org/issues/49/14.html.

[118] D. Ray and J. Ligatti, “Defining Code-Injection Attacks,” *SIGPLAN Notices*, vol. 47, no. 1, pp. 179–190, Jan. 2012, ISSN: 0362-1340. DOI: 10.1145/2103621.2103678. [Online]. Available: https://doi.org/10.1145/2103621.2103678.

[119] J. Forristal, “NT Web Technology Vulnerabilities,” *Phrack*, vol. 8, no. 54, Dec. 1998.

[120] W. G. Halfond, J. Viegas, and A. Orso, “A classification of SQL-injection attacks and countermeasures,” in *Proceedings of the IEEE international symposium on secure software engineering*, IEEE, vol. 1, 2006, pp. 13–15.

[121] M. Horner and T. Hyslip, “SQL Injection: The Longest Running Sequel in Programming History,” *The Journal of Digital Forensics , Security and Law*, vol. 12, no. 2, pp. 97–108, Jun. 2017.

[122] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman, “The Matter of Heartbleed,” in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC ’14, Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 475–488, ISBN: 9781450332132. DOI: 10.1145/2663716.2663755. [Online]. Available: https://doi.org/10.1145/2663716.2663755.

[123] National Institute of Standards and Technology, “CVE-2021-44228 Detail,” National Vulnerability Database, Tech. Rep. CVE-2021-44228, Dec. 2021. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-44228.

[124] 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, 2016, pp. 129–140, ISBN: 978-3-319-70578-1.

[125] Department for Digital, Culture, Media & Sport, “Code of Practice for Consumer IoT Security,” HM Government (UK), Tech. Rep., Oct. 2018. [Online]. Available: https://www.gov.uk/government/publications/code-of-practice-for-consumer-iot-security.

[126] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, “Understanding the Mirai Botnet,” in *Proceedings of the 26th USENIX Security Symposium*, Aug. 2017. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-antonakakis.pdf.

[127] T. Hunt. (Feb. 2016). “Controlling vehicle features of Nissan LEAFs across the globe via vulnerable APIs,” [Online]. Available: http://www.troyhunt.com/2016/02/controlling-vehicle-features-of-nissan.html (visited on 03/22/2021).

[128] D’Orazio, Christian J., Choo, Kim-Kwang Raymond, and Yang, Laurence T., “Data Exfiltration From Internet of Things Devices: iOS Devices as Case Studies,” *IEEE Internet of Things Journal*, vol. 4, no. 2, pp. 524–535, May 2016. DOI: 10.1109/JIOT.2016.2569094. [Online]. Available: http://ieeexplore.ieee.org/document/7470257/.

[129] B. Vignau, R. Khoury, and S. Hallé, “10 Years of IoT Malware: A Feature-Based Taxonomy,” in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2019, pp. 458–465. DOI: 10.1109/QRS-C.2019.00088.

[130] E. Ronen, A. Shamir, A.-O. Weingarten, and C. O’Flynn, “IoT Goes Nuclear: Creating a ZigBee Chain Reaction,” in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 195–212. DOI: 10.1109/SP.2017.14.

[131] 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.

[132] S. R. Zahra and M. Ahsan Chishti, “RansomWare and Internet of Things: A New Security Nightmare,” in *2019 9th International Conference on Cloud Computing, Data Science Engineering (Confluence)*, 2019, pp. 551–555. DOI: 10.1109/CONFLUENCE.2019.8776926.

[133] P. Bajpai and R. Enbody, “Preparing Smart Cities for Ransomware Attacks,” in *2020 3rd International Conference on Data Intelligence and Security (ICDIS)*, 2020, pp. 127–133. DOI: 10.1109/ICDIS50059.2020.00023.

[134] C. D. McDermott, F. Majdani, and A. V. Petrovski, “Botnet Detection in the Internet of Things using Deep Learning Approaches,” in *2018 International Joint Conference on Neural Networks (IJCNN)*, 2018, pp. 1–8. DOI: 10.1109/IJCNN.2018.8489489.

[135] R. Hallman, J. Bryan, G. Palavicini, J. Divita, and J. Romero-Mariona, “IoDDoS — The Internet of Distributed Denial of Service Attacks - A Case Study of the Mirai Malware and IoT-Based Botnets,” in *Proceedings of the 2nd International Conference on Internet of Things, Big Data and Security*, 2017, pp. 47–58, ISBN: 978-989-758-245-5. DOI: 10.5220/0006246600470058. [Online]. Available: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006246600470058.

[136] P. Syverson, “A taxonomy of replay attacks [cryptographic protocols],” in *Proceedings The Computer Security Foundations Workshop VII*, 1994, pp. 187–191. DOI: 10.1109/CSFW.1994.315935.

[137] D. Kushner, “The real story of Stuxnet,” *IEEE Spectrum*, pp. 48–53, Mar. 2013. DOI: 10.1109/MSPEC.2013.6471059. [Online]. Available: http://ieeexplore.ieee.org/document/6471059/.

[138] R. Langner, “Stuxnet: Dissecting a Cyberwarfare Weapon,” *IEEE Security Privacy*, vol. 9, no. 3, pp. 49–51, 2011. DOI: 10.1109/MSP.2011.67.

[139] D. Albright, P. Brannan, and C. Walrond, “Stuxnet Malware and Natanz,” Institute for Science and International Security, Tech. Rep., Feb. 2011. [Online]. Available: http://isis-online.org/isis-reports/detail/stuxnet-malware-and-natanz-update-of-isis-december-22-2010-reportsupa-href1/8.

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

[141] B. Huang, A. A. Cardenas, and R. Baldick, “Not everything is dark and gloomy: Power grid protections against IoT demand attacks,” in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1115–1132.

[142] D. Palmer, S. Fazzari, and S. Wartenberg, “Defense Systems and IoT: Security Issues in an Era of Distributed Command and Control,” in *Proceedings of the 26th Edition on Great Lakes Symposium on VLSI*, ser. GLSVLSI ’16, Boston, Massachusetts, USA: Association for Computing Machinery, 2016, pp. 175–179, ISBN: 9781450342742. DOI: 10.1145/2902961.2903038. [Online]. Available: https://doi.org/10.1145/2902961.2903038.

[143] R. E. Hiromoto, M. Haney, and A. Vakanski, “A secure architecture for IoT with supply chain risk management,” in *2017 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 1, 2017, pp. 431–435. DOI: 10.1109/IDAACS.2017.8095118.

[144] M. Ohm, H. Plate, A. Sykosch, and M. Meier, “Backstabber’s Knife Collection: A Review of Open Source Software Supply Chain Attacks,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Maurice, L. Bilge, G. Stringhini, and N. Neves, Eds., Cham: Springer International Publishing, 2020, pp. 23–43, ISBN: 978-3-030-52683-2.

[145] D.-L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, “Typosquatting and Combosquatting Attacks on the Python Ecosystem,” in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, 2020, pp. 509–514. DOI: 10.1109/EuroSPW51379.2020.00074.

[146] National Cyber Security Center of the Netherlands, “Cyber Security Assessment Netherlands 2017,” National Cyber Security Center of the Netherlands, Tech. Rep. CSAN2017, Aug. 2017. [Online]. Available: https://www.ncsc.nl/english/current-topics/Cyber+Security+Assessment+Netherlands/cyber-security-assessment-netherlands-2017.html.

[147] W. A. Arbaugh, W. L. Fithen, and J. McHugh, “Windows of vulnerability: A case study analysis,” *Computer*, vol. 33, no. 12, pp. 52–59, 2000. DOI: 10.1109/2.889093.

[148] S. Datta Burton, L. M. Tanczer, S. Vasudevan, S. Hailes, and M. Carr, “The UK Code of Practice for Consumer IoT Cybersecurity: ’where we are and what next’.,” Mar. 2021. DOI: DOI:10.14324/000.rp.10117734.

[149] D. Kahn, *The Code Breakers: The Compreshnsive History of Secret Communication fropm Ancient Times to the Internet*. Scribner, 1996.

[150] F. W. Winterbotham, *The Ultra Secret*. London: Weidenfeld and Nicolson, Oct. 1974, ISBN: 0297768328.

[151] G. Welchman, *The Hut Siz Story*. McGraw-Hill, Mar. 1982, ISBN: 0070691800.

[152] J. H. Ellis, “The possibility of Secure “Non-Secret” Digital Encryption,” GCHQ, Tech. Rep. 3006, Jan. 1970.

[153] C. C. Cocks, “A Note on ‘Non-Secret Encryption’,” GCHQ, Tech. Rep., Nov. 1973.

[154] R. L. Rivest, A. Shamir, and L. M. Adleman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.,” *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978. DOI: 10.1145/359340.359342.

[155] V. S. Miller, “Use of elliptic curves in cryptography,” in *Conference on the theory and application of cryptographic techniques*, Springer, 1985, pp. 417–426.

[156] N. Koblitz, “Elliptic curve cryptosystems,” *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, 1987.

[157] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[158] 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.

[159] US-CERT, “SSL 3.0 Protocol Vulnerability and POODLE Attack,” Tech. Rep. Alert (TA14-290A), Oct. 2014. [Online]. Available: https://www.us-cert.gov/ncas/alerts/TA14-290A.

[160] T. Dierks and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2,” Internet Engineering Task Force, Tech. Rep. RFC5246, Aug. 2008. DOI: 10.17487/rfc5246. [Online]. Available: https://www.rfc-editor.org/info/rfc5246.

[161] W. Diffie and M. Hellman, “New directions in cryptography,” *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976. DOI: 10.1109/TIT.1976.1055638.

[162] C. Cheng, R. Lu, A. Petzoldt, and T. Takagi, “Securing the Internet of Things in a Quantum World,” *IEEE Communications Magazine*, vol. 55, no. 2, pp. 116–120, 2017. DOI: 10.1109/MCOM.2017.1600522CM.

[163] A. Steane, “Quantum computing,” *Reports on Progress in Physics*, vol. 61, no. 2, p. 117, 1998.

[164] P. W. Shor, “Algorithms for quantum computation: Discrete logarithms and factoring,” in *Proceedings 35th annual symposium on foundations of computer science*, Ieee, 1994, pp. 124–134.

[165] T. Monz, D. Nigg, E. A. Martinez, M. F. Brandl, P. Schindler, R. Rines, S. X. Wang, I. L. Chuang, and R. Blatt, “Realization of a scalable Shor algorithm,” *Science*, vol. 351, no. 6277, pp. 1068–1070, 2016.

[166] U. Pujeri, P. S. Aithal, and R. Pujeri, “Survey of Lattice to Design Post Quantum Cryptographic Algorithm Using Lattice,” *International Journal of Engineering Trends and Technology*, vol. 69, no. 1, pp. 92–96, Jan. 2021, (January 25, 2021). DOI: 10.2139/ssrn.3805387.

[167] R. Braden, “Requirements for Internet Hosts – Communication Layers,” Tech. Rep. RFC 1122, Oct. 1, 1989. [Online]. Available: https://tools.ietf.org/html/rfc1122.

[168] IEEE, *IEEE Standard for Ethernet*, IEEE, Piscataway, NJ, USA, Mar. 2016. DOI: 10.1109/IEEESTD.2016.7428776. [Online]. Available: http://ieeexplore.ieee.org/document/7428776/.

[169] ——, *IEEE Standard for Wireless LAN*, IEEE, Piscataway, NJ, USA, Dec. 2016. DOI: 10.1109/IEEESTD.2016.7786995. [Online]. Available: http://ieeexplore.ieee.org/document/7786995/.

[170] N. Abramson, “Development of the ALOHANET,” *IEEE Transactions on Information Theory*, vol. 31, no. 2, pp. 119–123, 1985. DOI: 10.1109/TIT.1985.1057021.

[171] C. E. Spurgeon and J. Zimmerman, *Ethernet: the definitive guide*, Second. O'Reilly Media, Inc., Mar. 2014, ISBN: 978-1-449-36184-6.

[172] Xerox Corporation, *ALTO: A Personal Computer System Hardware Manual*, Xerox Corporation, 3333 Coyote Hill Road, Palo Alto, California, Aug. 1976. [Online]. Available: http://www.bitsavers.org/pdf/xerox/alto/Alto_Hardware_Manual_Aug76.pdf.

[173] P. C. Jain, "Recent trends in next generation terabit Ethernet and gigabit wireless local area network," in *2016 International Conference on Signal Processing and Communication (ICSC)*, 2016, pp. 106–110. DOI: 10.1109/ICSPCom.2016.7980557.

[174] IEEE, "IEEE Standard for Ethernet Amendment 10: Media Access Control Parameters, Physical Layers, and Management Parameters for 200 Gb/s and 400 Gb/s Operation," IEEE, Tech. Rep. IEEE 802.3bs-2017, Dec. 2017. [Online]. Available: https://standards.ieee.org/standard/802_3bs-2017.html.

[175] V. Cerf and R. Kahn, "A Protocol for Packet Network Intercommunication," *IEEE Transactions on Communications*, vol. 22, no. 5, pp. 637–648, 1974. DOI: 10.1109/TCOM.1974.1092259. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1092259.

[176] B. M. Leiner, V. G. Cerf, D. D. Clark, R. E. Kahn, L. Kleinrock, D. C. Lynch, J. Postel, L. G. Roberts, and S. S. Wolff, "The Past and Future History of the Internet," *Communications of the ACM*, vol. 40, no. 2, pp. 102–108, Feb. 1997, ISSN: 0001-0782. DOI: 10.1145/253671.253741. [Online]. Available: https://doi.org/10.1145/253671.253741.

[177] S. McKenzie, "Asynchronous Transfer Mode (ATM)," in *Encyclopedia of Computer Science*. GBR: John Wiley and Sons Ltd., 2003, pp. 107–108, ISBN: 0470864125.

[178] J. Postel, "Transmission Control Protocol," Internet Engineering Task Force, Tech. Rep. RFC793, Sep. 1981. DOI: 10.17487/rfc0793. [Online]. Available: https://tools.ietf.org/html/rfc793.

[179] J. Postel, "User Datagram Protocol," Internet Engineering Task Force, Tech. Rep. RFC768, Aug. 1980. DOI: 10.17487/rfc0768. [Online]. Available: https://www.rfc-editor.org/info/rfc0768.

[180] F. X. Anklesaria, M. P. McCahill, P. Lindner, D. Johnson, D. Torrey, and B. Albert, "The Internet Gopher Protocol (a distributed document search and retrieval protocol).," Internet Engineering Task Force, Tech. Rep. RFC1436, 1993. DOI: 10.17487/RFC1436. [Online]. Available: https://www.rfc-editor.org/info/rfc1436.

[181] T. Berners-Lee, "Information Management: A Proposal," Conseil Européen pour la Recherche Nucléaire, Geneva, Tech. Rep., 1990. [Online]. Available: https://www.w3.org/History/1989/proposal.html.

[182] National Institute of Standards and Technology, “Announcing the Advanced Encryption Standard (AES),” National Institute of Standards and Technology, Tech. Rep. FIPS197, Nov. 2001. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[183] J. Demen and V. Rijmen, “AES Proposal: Rijndael,” National Institute of Standards and Technology, Tech. Rep., 1999.

[184] W. Penard and T. van Werkhoven, “On the Secure Hash Algorithm family,” National Institute of Standards and Technology, Tech. Rep., 2001.

[185] National Institute of Standards and Technology, “Announcing Approval of Federal Information Processing Standard (FIPS) 180-2, Secure Hash Standard,” National Institute of Standards and Technology, Tech. Rep. FIPS180-2, 2002.

[186] K. Seo and S. Kent, “Security Architecture for the Internet Protocol,” Tech. Rep. RFC4301, Dec. 2005. [Online]. Available: http://tools.ietf.org/html/rfc4301.

[187] S. Raza, S. Duquennoy, J. Höglund, U. Roedig, and T. Voigt, “Secure communication for the Internet of Things—a comparison of link-layer security and IPsec for 6LoWPAN,” *Security and Communication Networks*, vol. 7, no. 12, pp. 2654–2668, Dec. 2014. DOI: 10.1002/sec.406. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/sec.406/full.

[188] T. Dierks and C. Allen, “The TLS Protocol Version 1.0,” Tech. Rep. RFC2246, Jan. 1999. [Online]. Available: https://tools.ietf.org/html/rfc2246.

[189] A. Freier, P. Karlton, and P. Kocher, “The Secure Sockets Layer (SSL) Protocol Version 3.0,” Internet Engineering Task Force, Tech. Rep. RFC6101, Aug. 2011. [Online]. Available: https://tools.ietf.org/html/rfc6101.

[190] E. Rescorla, “HTTP over TLS,” Internet Engineering Task Force, Tech. Rep. RFC2818, May 2000. [Online]. Available: https://tools.ietf.org/html/rfc2818.

[191] Let’s Encrypt. (). “About Let’s Encrypt,” [Online]. Available: https://letsencrypt.org/about/.

[192] International Telecommunication Union, “X.509 : Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks,” International Telecommunications Union, Tech. Rep. X.509 (10/19), Oct. 2019.

[193] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile,” Internet Engineering Task Force, Tech. Rep., 2008.

[194] J. Roskind, “QUIC: Design Document and Specification Rationale,” Google, Tech. Rep., Apr. 2012. [Online]. Available: https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/.

[195] P. Biswal and O. Gnawali, “Does QUIC Make the Web Faster?” In *2016 IEEE Global Communications Conference (GLOBECOM)*, Dec. 2016, pp. 1–6. DOI: 10.1109/GLOCOM.2016.7841749.

[196] Y. Cui, T. Li, C. Liu, X. Wang, and M. Kühlewind, “Innovating Transport with QUIC: Design Approaches and Research Challenges,” *IEEE Internet Computing*, vol. 21, no. 2, pp. 72–76, Mar. 2017, ISSN: 1089-7801. DOI: 10.1109/MIC.2017.44.

[197] D. Murray, T. Koziniec, S. Zander, M. Dixon, and P. Koutsakis, “An analysis of changing enterprise network traffic characteristics,” in *2017 23rd Asia-Pacific Conference on Communications (APCC)*, Dec. 2017, pp. 1–6. DOI: 10.23919/APCC.2017.8303960.

[198] E. Rescorla and N. Modadugu, “Datagram Transport Layer Security Version 1.2,” Internet Engineering Task Force, Tech. Rep., 2012.

[199] T. Kothmayr, C. Schmitt, W. Hu, M. Brünig, and G. Carle, “DTLS based security and two-way authentication for the Internet of Things,” *Ad Hoc Networks*, vol. 11, no. 8, pp. 2710–2723, 2013, ISSN: 1570-8705. DOI: https://doi.org/10.1016/j.adhoc.2013.05.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570870513001029.

[200] N. AlFardan and K. G. Paterson, “Plaintext-recovery attacks against datagram TLS,” in *Network and Dstributed System Security symposium (NDSS 2012)*, 2012.

[201] T. Ylönen, “The Secure Shell (SSH) Connection Protocol,” Internet Engineering Task Force, Tech. Rep. RFC 4254, 2006. [Online]. Available: https://tools.ietf.org/html/rfc4254.

[202] R. T. Fielding and R. N. Taylor, “Principled design of the modern Web architecture,” in *International Conference on Software Engineering*, New York, New York, USA: ACM Press, 2000, pp. 407–416. DOI: 10.1145/337180.337228. [Online]. Available: http://dx.doi.org/10.1145%2F337180.337228.

[203] N. Naik, “Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP,” in *2017 IEEE International Systems Engineering Symposium (ISSE)*, Oct. 2017, pp. 1–7. DOI: 10.1109/SysEng.2017.8088251.

[204] H. W. van der Westhuizen and G. P. Hancke, “Practical Comparison between COAP and MQTT - Sensor to Server level,” in *2018 Wireless Advanced (WiAd)*, 2018, pp. 1–6. DOI: 10.1109/WIAD.2018.8588443.

[205] Y. Guamán, G. Ninahualpa, G. Salazar, and T. Guarda, “Comparative Performance Analysis between MQTT and CoAP Protocols for IoT with Raspberry PI 3 in IEEE 802.11 Environments,” in *2020 15th Iberian Conference on Information Systems and Technologies (CISTI)*, 2020, pp. 1–6. DOI: 10.23919/CISTI49556.2020.9140905.

[206] C. Bormann, A. P. Castellani, and Z. Shelby, “CoAP: An Application Protocol for Billions of Tiny Internet Nodes,” *IEEE Internet Computing*, vol. 16, no. 2, pp. 62–67, 2012. DOI: 10.1109/MIC.2012.29.

[207] A. Rahman and E. Dijk, “Group Communication for the Constrained Application Protocol (CoAP),” Tech. Rep. RFC7390, Oct. 2014. [Online]. Available: https://tools.ietf.org/html/rfc7390.

[208] Z. Shelby, “Constrained RESTful Environments (CoRE) Link Format,” Tech. Rep. RFC6690, Aug. 2012. [Online]. Available: https://tools.ietf.org/html/rfc6690.

[209] R. A. Rahman and B. Shah, “Security analysis of IoT protocols: A focus in CoAP,” in *2016 3rd MEC International Conference on Big Data and Smart City (ICBDSC)*, 2016, pp. 1–7. DOI: 10.1109/ICBDSC.2016.7460363.

[210] G. Tanganelli, C. Vallati, and E. Mingozzi, “CoAPthon: Easy development of CoAP-based IoT applications with Python,” in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, 2015, pp. 63–68. DOI: 10.1109/WF-IoT.2015.7389028.

[211] A. Banks, E. Briggs, K. Borgendale, and R. Gupta, “MQTT Specification (version 5.0),” OASIS, Tech. Rep., Mar. 2019. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf.

[212] R. A. Light, “Mosquitto: server and client implementation of the MQTT protocol,” *Journal of Open Source Software*, vol. 2, no. 13, p. 265, 2017.

[213] K. Hwang, J. M. Lee, I. H. Jung, and D. Lee, “Modification of Mosquitto Broker for Delivery of Urgent MQTT Message,” in *2019 IEEE Eurasia Conference on IOT, Communication and Engineering (ECICE)*, 2019, pp. 166–167. DOI: 10.1109/ECICE47484.2019.8942800.

[214] P. Eronen and H. Tschofenig, “Pre-shared key ciphersuites for transport layer security (TLS),” Tech. Rep. RFC4048, 2005. DOI: 10.17487/rfc4048. [Online]. Available: https://www.rfc-editor.org/info/rfc4048.

[215] S. Han, E. Chang, W. Liu, J. Wang, and V. Potdar, “A new encryption algorithm over elliptic curve,” in *INDIN ’05. 2005 3rd IEEE International Conference on Industrial Informatics, 2005.*, 2005, pp. 675–679. DOI: 10.1109/INDIN.2005.1560456.

[216] North Atlantic Treaty Organization, *Allied Joint Dctrine for the conduct of operations*, AJP-3(B). NATO, Jun. 2017. [Online]. Available: https://www.gov.uk/government/publications/allied-joint-doctrine-for-the-conduct-of-operations-ajp-3b.

[217] W. Bai, M. Pearson, P. G. Kelley, and M. L. Mazurek, “Improving Non-Experts’ Understanding of End-to-End Encryption: An Exploratory Study,” in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, 2020, pp. 210–219. DOI: 10.1109/EuroSPW51379.2020.00036.

[218] M. van Straten, “The Google Cloud powers your Philips Hue Lightbulbs,” in *GDG DevFest Ukraine 2018*, 2018. [Online]. Available: https://speakerdeck.com/crunchie84/the-google-cloud-powers-your-philips-hue-lightbulbs.

[219] A. Kott, “Challenges and characteristics of intelligent autonomy for Internet of Battle Things in highly adversarial environments,” *arXiv preprint arXiv:1803.11256*, 2018.

[220] A. Raglin, S. Metu, S. Russell, and P. Budulas, “Implementing Internet of Things in a military command and control environment,” *Proceedings of the Society of Photo-Optical Instrumentation Engineers*, vol. 10207, May 2017. DOI: 10.1117/12.2265030. [Online]. Available: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/10207/1020708/Implementing-Internet-of-Things-in-a-military-command-and-control/10.1117/12.2265030.full.

[221] A. Kott and D. S. Alberts, “How Do You Command an Army of Intelligent Things?” *IEEE Computer*, vol. 50, no. 12, pp. 96–100, Dec. 2017. DOI: 10.1109/MC.2017.4451205.

[222] S. Samonas and D. Coss, “The CIA Strikes Back: Redefinning Confidentiality, Integrity and Availability in security.,” *Journal of Information System Security*, vol. 10, no. 3, 2014.

[223] J. Braun, J. Buchmann, D. Demirel, M. Geihs, M. Fujiwara, S. Moriai, M. Sasaki, and A. Waseda, “LINCOS: A Storage System Providing Long-Term Integrity, Authenticity, and Confidentiality,” in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS ’17, Abu Dhabi, United Arab Emirates: Association for Computing Machinery, 2017, pp. 461–468, ISBN: 9781450349444. DOI: 10.1145/3052973.3053043. [Online]. Available: https://doi.org/10.1145/3052973.3053043.

[224] K. Arakadakis, P. Charalampidis, A. Makrogiannakis, and A. Fragkiadakis, *Firmware over-the-air programming techniques for iot networks – a survey*, 2020. arXiv: 2009.02260 [cs.NI].

[225] J. L. Hernández-Ramos, G. Baldini, S. N. Matheu, and A. Skarmeta, “Updating IoT devices: challenges and potential approaches,” in *2020 Global Internet of Things Summit (GIoTS)*, 2020, pp. 1–5. DOI: 10.1109/GIOTS49054.2020.9119514.

[226] O. Grote, A. Ahrens, and C. Benavente-Peces, “A Review of Post-quantum Cryptography and Crypto-agility Strategies,” in *2019 International Interdisciplinary PhD Workshop (IIPhDW)*, 2019, pp. 115–120. DOI: 10.1109/IIPHDW.2019.8755433.

[227] D. Peng, L. Cao, and W. Xu, “Using JSON for data exchanging in web service applications,” *Journal of Computational Information Systems*, vol. 7, no. 16, pp. 5883–5890, 2011.

[228] K. Maeda, "Performance evaluation of object serialization libraries in XML, JSON and binary formats," in *2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)*, 2012, pp. 177–182. DOI: 10.1109/DICTAP.2012.6215346.

[229] S. Wen and W. Dang, "Research on Base64 Encoding Algorithm and PHP Implementation," in *2018 26th International Conference on Geoinformatics*, 2018, pp. 1–5. DOI: 10.1109/GEOINFORMATICS.2018.8557068.

[230] S. Popić, D. Pezer, B. Mrazovac, and N. Teslić, "Performance evaluation of using Protocol Buffers in the Internet of Things communication," in *2016 International Conference on Smart Systems and Technologies (SST)*, 2016, pp. 261–265. DOI: 10.1109/SST.2016.7765670.

[231] J. Jonsson and B. Kaliski, "Public-key cryptography standards (PKCS)# 1: RSA cryptography specifications version 2.1," Internet Engineering Task Force (IETF), Tech. Rep. RFC 3447, Feb. 2003.

[232] S. Furuhashi. (). "MessagePack," [Online]. Available: https://msgpack.org/index.html (visited on 04/04/2018).

[233] F. Van den Abeele, J. Haxhibeqiri, I. Moerman, and J. Hoebeke, "Scalability Analysis of Large-Scale LoRaWAN Networks in ns-3," *IEEE Internet of Things Journal*, vol. 4, no. 6, pp. 2186–2198, 2017. DOI: 10.1109/JIOT.2017.2768498.

[234] P. Leach, M. Mealling, and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace," RFC Editor, Tech. Rep. RFC4122, Jul. 2005. DOI: 10.17487/rfc4122.

[235] B. Schiling, *The Boost C++ Libraries*. XML Press, Jul. 2011, ISBN: 0982219199. [Online]. Available: http://www.worldcat.org/title/boost-c-libraries/oclc/929660203.

[236] F. H. Mathis, "A Generalized Birthday Problem," *SIAM Review*, vol. 33, no. 2, pp. 265–270, Jul. 1991. DOI: 10.1137/1033051. [Online]. Available: https://doi.org/10.1137/1033051.

[237] S. Delaune, S. Kremer, M. D. Ryan, and G. Steel, "A Formal Analysis of Authentication in the TPM," in *Formal Aspects of Security and Trust*, P. Degano, S. Etalle, and J. Guttman, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 111–125, ISBN: 978-3-642-19751-2.

[238] S. L. Kinney, *Trusted platform module basics: using TPM in embedded systems*. Elsevier, 2006.

[239] G. E. Suh and S. Devadas, "Physical Unclonable Functions for Device Authentication and Secret Key Generation," in *2007 44th ACM/IEEE Design Automation Conference*, Jun. 2007, pp. 9–14.

[240] M. Sethi, J. Arkko, A. Keranen, and H. Back, "Practical Considerations and Implementation Experiences in Securing Smart Object Networks," Internet Engineering Task Force, Tech. Rep. draft-ietf-lwig-crypto-sensors-05, Dec. 2017. [Online]. Available: https://tools.ietf.org/html/draft-ietf-lwig-crypto-sensors-05.

[241] S. Garfinkel, “An Evaluation of Amazon’s Grid Computing Services: EC2, S3, and SQS,” Harvard University Computer Science Group, Tech. Rep. TR-08-07, 2007. [Online]. Available: http://nrs.harvard.edu/urn-3:HUL.InstRepos:24829568.

[242] S. Tiwari, “An Introduction to QR Code Technology,” in *2016 International Conference on Information Technology (ICIT)*, 2016, pp. 39–44. DOI: 10.1109/ICIT.2016.021.

[243] G. Madlmayr, J. Langer, C. Kantner, and J. Scharinger, “NFC Devices: Security and Privacy,” in *2008 Third International Conference on Availability, Reliability and Security*, 2008, pp. 642–647. DOI: 10.1109/ARES.2008.105.

[244] K. E. Jeon, J. She, P. Soonsawad, and P. C. Ng, “BLE Beacons for Internet of Things Applications: Survey, Challenges, and Opportunities,” *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 811–828, 2018. DOI: 10.1109/JIOT.2017.2788449.

[245] D. Bhattacharya, M. Canul, and S. Knight, “Impact of the Physical Web and BLE Beacons,” in *Proceedings of the 5th Annual Conference on Research in Information Technology*, ser. RIIT ’16, Boston, Massachusetts, USA: Association for Computing Machinery, 2016, p. 53, ISBN: 9781450344531. DOI: 10.1145/2978178.2978179. [Online]. Available: https://doi.org/10.1145/2978178.2978179.

[246] C. Henderson, “IoT: End of Shorter Days,” in *RSA Conference 2017*, San Francisco, 2017. [Online]. Available: https://www.rsaconference.com/videos/iot-end-of-shorter-days.

[247] S. Josefsson, “The Base16, Base32, and Base64 Data Encodings,” Internet Engineering Task Force (IETF), Tech. Rep. RFC3548, Jul. 2003.

[248] J. Tan, L. Bauer, J. Bonneau, L. F. Cranor, J. Thomas, and B. Ur, “Can Unicorns Help Users Compare Crypto Key Fingerprints?” In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, ser. CHI ’17, New York, NY, USA: Association for Computing Machinery, 2017, pp. 3787–3798, ISBN: 9781450346559. DOI: 10.1145/3025453.3025733. [Online]. Available: https://doi.org/10.1145/3025453.3025733.

[249] N. Gordon, “Colour blindness,” *Public Health*, vol. 112, no. 2, pp. 81–84, 1998, ISSN: 0033-3506. DOI: https://doi.org/10.1038/sj.ph.1900446. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0033350698005903.

[250] D. L. McDonald, “A convention for human-readable 128-bit keys,” Internet Engineering Task Force, Tech. Rep. RFC 1751, 1994. DOI: 10.17487/rfc1751. [Online]. Available: https://www.rfc-editor.org/info/rfc1751.

[251] M. D. Leonhard and V. N. Venkatakrishnan, “A comparative study of three random password generators,” in *2007 IEEE International Conference on Electro/Information Technology*, 2007, pp. 227–232. DOI: 10.1109/EIT.2007.4374533.

[252] Y. Chen, J. Au, P. Kazlas, A. Ritenour, H. Gates, and M. McCreary, “Flexible active-matrix electronic ink display,” *Nature*, vol. 423, no. 69366936, pp. 136–136, May 2003, ISSN: 1476-4687. DOI: 10.1038/423136a.

[253] A. J. Poulter, *pySRUP – Human verficiation of device identity*, Video, Sep. 2019. [Online]. Available: https://youtu.be/-qBzZ3wT1Tc.

[254] O. Lobachev, “Direct visualization of cryptographic keys for enhanced security,” *The Visual Computer*, vol. 34, no. 12, pp. 1749–1759, Dec. 2018. DOI: 10.1007/s00371-017-1466-6.

[255] P. R. Zimmermann, *The official PGP user’s guide*. MIT press, 1995.

[256] D. Loss, T. Limmer, and A. von Gernler, “The Drunken Bishop: An Analysis of the OpenSSH Fingerprint Visualization Algorithm,” Jan. 2009.

[257] M. Naor and A. Shamir, “Visual cryptography,” in *Advances in Cryptology — EUROCRYPT’94*, A. De Santis, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 1–12, ISBN: 978-3-540-44717-7. DOI: 10.1007/BFb0053418. [Online]. Available: http://link.springer.com/10.1007/BFb0053418.

[258] International Organization for Standardization, “Information technology — Automatic identification and data capture techniques — Code 128 bar code symbology specification,” International Organization for Standardization, Tech. Rep. ISO/IEC 15417:2007, Jun. 2007. [Online]. Available: https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/04/38/43896.html.

[259] I.-C. Dita, M. Otesteanu, and F. Quint, “Data Matrix Code — A reliable optical identification of microelectronic components,” in *2011 IEEE 17th International Symposium for Design and Technology in Electronic Packaging (SIITME)*, 2011, pp. 39–44. DOI: 10.1109/SIITME.2011.6102683.

[260] R. Kulshreshtha, A. Kamboj, and S. Singh, “Decoding robustness performance comparison for QR and data matrix code,” ser. CCSEIT ’12, Association for Computing Machinery, Oct. 2012, pp. 722–731, ISBN: 978-1-4503-1310-0. DOI: 10.1145/2393216.2393337. [Online]. Available: https://doi.org/10.1145/2393216.2393337.

[261] L. Hudson, V. Kursancew, and J. Weston, *pylibdmtx*, https://github.com/NaturalHistoryMuseum/pylibdmtx, May 2020.

[262] L. Hudson and A. Newby, *pyzbar*, https://github.com/NaturalHistoryMuseum/pyzbar, Natural History Museum, May 2020.

[263] J. C. Haartsen, “The Bluetooth radio system,” *IEEE Personal Communications*, vol. 7, no. 1, pp. 28–36, 2000. DOI: 10.1109/98.824570.

[264] S. Safaric and K. Malaric, “ZigBee wireless standard,” in *Proceedings of 2006 International Symposium on Electronics in Marine (ELMAR 2006)*, M. Grgić and S. Grgić, Eds., IEEE, Zadar, Croatia: Institute of Electrical and Electronic Engineers, Jun. 2006, pp. 259–262. DOI: 10.1109/ELMAR.2006.329562.

[265] S. A. Weis, “RFID (radio frequency identification): Principles and applications,” *System*, vol. 2, no. 3, pp. 1–23, 2007.

[266] F. D. Garcia, G. de Koning Gans, R. Muijrers, P. Van Rossum, R. Verdult, R. W. Schreur, and B. Jacobs, “Dismantling MIFARE classic,” in *European symposium on research in computer security*, 2008, pp. 97–114.

[267] NFC Forum, *NFC Data Exchange Format (NDEF) Technical Specification*, Jul. 2006.

[268] NFC Forum Technical Specification, *SNEP: Simple NDEF Exchange Protcol*. [Online]. Available: http://www.nfcforum.org/specs/spec-list/.

[269] A. Lotito and D. Mazzocchi, “OPEN-SNEP Project: Enabling P2P over NFC Using NPP and SNEP,” in *2013 5th International Workshop on Near Field Communication (NFC)*, Feb. 2013, pp. 1–6. DOI: 10.1109/NFC.2013.6482447.

[270] NXP Semiconductors B.V., *PN532/C1 - Near Field Communication (NFC) controller*, NXP B.V., Nov. 2017. [Online]. Available: https://www.nxp.com/docs/en/nxp/data-sheets/PN532_C1.pdf.

[271] Microchip Technology Inc., *ATmega16U4 / ATmega32U4 — 8-bit Microcontroller with 16/32K bytes of ISP Flash and USB Controller*, Microchip Technology Inc., 2016. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7766-8-bit-AVR-ATmega16U4-32U4_Summary.pdf.

[272] A. J. Poulter, *SRUP Machine Moderated Joins*, Video, Aug. 2020. [Online]. Available: https://youtu.be/Vi135raj1LE.

[273] M. El-hajj, A. Fadlallah, M. Chamoun, and A. Serhrouchni, “A Survey of Internet of Things (IoT) Authentication Schemes,” *Sensors*, vol. 19, no. 5, p. 1141, Jan. 2019. DOI: 10.3390/s19051141.

[274] J. Bradley, W. Denniss, H. Tschofenig, and M. Jones, *OAuth 2.0 Device Authorization Grant*, en, https://tools.ietf.org/html/rfc8628, Aug. 2019.

[275] D. Hardt, “The OAuth 2.0 Authorization Framework,” Tech. Rep. RFC6749, 2012. DOI: 10.17487/rfc6749. [Online]. Available: https://www.rfc-editor.org/info/rfc6749.

[276] J. Franks, P. J. Leach, A. Luotonen, P. M. Hallam-Baker, S. D. Lawrence, J. L. Hostetler, and L. C. Stewart, “HTTP Authentication: Basic and Digest Access Authentication,” Tech. Rep. RFC2617, Jun. 1999. DOI: 10.17487/rfc2617. [Online]. Available: https://www.rfc-editor.org/info/rfc2617.

[277] J. Reschke, “The ’Basic’ HTTP Authentication Scheme,” Internet Engineering Task Force, Tech. Rep. RFC7617, Sep. 2015. [Online]. Available: https://tools.ietf.org/html/rfc7617.

[278] Y. Feng, W. Wang, Y. Weng, and H. Zhang, "A Replay-Attack Resistant Authentication Scheme for the Internet of Things," in *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, vol. 1, Jul. 2017, pp. 541–547. DOI: 10.1109/CSE-EUC.2017.101.

[279] K. Greene, D. Rodgers, H. Dykhuizen, K. McNeil, Q. Niyaz, and K. A. Shamaileh, "Timestamp-based Defense Mechanism Against Replay Attack in Remote Keyless Entry Systems," in *2020 IEEE International Conference on Consumer Electronics (ICCE)*, 2020, pp. 1–4. DOI: 10.1109/ICCE46568.2020.9043039.

[280] S. M. Bellovin and M. Merritt, "Limitations of the Kerberos Authentication System," *ACM SIGCOMM Computer Communication Review*, vol. 20, no. 5, pp. 119–132, Oct. 1990, ISSN: 0146-4833. DOI: 10.1145/381906.381946. [Online]. Available: https://doi.org/10.1145/381906.381946.

[281] D. Mills and U. Delaware, "Https://www.ietf.org/rfc/rfc5905.txt," Internet Engineering Task Force, Tech. Rep. RFC 5905, Jun. 2010. [Online]. Available: https://www.ietf.org/rfc/rfc5905.txt.

[282] K. H. M. Wong, Y. Zheng, J. Cao, and S. Wang, "A dynamic user authentication scheme for wireless sensor networks," in *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC'06)*, vol. 1, 2006. DOI: 10.1109/SUTC.2006.1636182.

[283] S. Tomasin, S. Zulian, and L. Vangelista, "Security Analysis of LoRaWAN Join Procedure for Internet of Things Networks," in *2017 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, 2017, pp. 1–6. DOI: 10.1109/WCNCW.2017.7919091.

[284] K. Pavani and P. Sriramya, "Enhancing Public Key Cryptography using RSA, RSA-CRT and N-Prime RSA with Multiple Keys," in *2021 Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV)*, 2021, pp. 1–6. DOI: 10.1109/ICICV50876.2021.9388621.

[285] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The First Collision for Full SHA-1," English, in *Advances in Cryptology*, Santa Barbara, CA: Springer, Cham, Aug. 2017, pp. 570–596, ISBN: 978-3-319-63687-0. DOI: 10.1007/978-3-319-63688-7_19. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-63688-7_19.

[286] H. Sochor, F. Ferrarotti, and R. Ramler, "Automated Security Test Generation for MQTT Using Attack Patterns," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ser. ARES '20, Virtual Event, Ireland: Association for Computing Machinery, 2020, ISBN: 9781450388337. DOI: 10.1145/3407023.3407078. [Online]. Available: https://doi.org/10.1145/3407023.3407078.

[287] S. N. Firdous, Z. Baig, C. Valli, and A. Ibrahim, “Modelling and Evaluation of Malicious Attacks against the IoT MQTT Protocol,” in *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, Jun. 2017, pp. 748–755. DOI: 10.1109/iThings-GreenCom-CPSCom-SmartData.2017.115.

[288] M. Bogdanoski, T. Shuminoski, and A. Risteski, “Analysis of the SYN Flood DoS Attack,” *International Journal of Computer Network and Information Security*, vol. 5, no. 8, pp. 15–11, Jun. 2013. DOI: 10.5815/ijcnis.2013.08.01. [Online]. Available: http://www.mecs-press.org/ijcnis/ijcnis-v5-n8/v5n8-1.html.

[289] S. N. Firdous, Z. Baig, A. Ibrahim, and C. Valli, “Denial of service attack detection through machine learning for the IoT,” *Journal of Information and Telecommunication*, vol. 4, no. 4, pp. 482–503, 2020. DOI: 10.1080/24751839.2020.1767484. eprint: https://doi.org/10.1080/24751839.2020.1767484. [Online]. Available: https://doi.org/10.1080/24751839.2020.1767484.

[290] M. M. Hafiz and F. H. Mohd Ali, “Profiling and mitigating brute force attack in home wireless LAN,” in *2014 International Conference on Computational Science and Technology (ICCST)*, 2014, pp. 1–6. DOI: 10.1109/ICCST.2014.7045190.

[291] S. J. Johnston, M. Scott, and S. J. Cox, “Recommendations for securing Internet of Things devices using commodity hardware,” in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, Dec. 2016, pp. 307–310. DOI: 10.1109/WF-IoT.2016.7845410.

[292] A. J. Poulter, *The Secure Remote Update Protocol v.6.0*, GitHub.com, Jun. 2021. DOI: 10.5281/zenodo.5041190. [Online]. Available: https://github.com/dstl/srup.

[293] B. Stroustrup, *The C++ Programming Language*, 4th Edition. Addison-Wesley Professional, May 2013, ISBN: 9780133522884.

[294] S. Koranne, “Boost C++ Libraries,” in *Handbook of Open Source Tools*. Boston, MA: Springer US, 2011, pp. 127–143, ISBN: 978-1-4419-7719-9. DOI: 10.1007/978-1-4419-7719-9_6. [Online]. Available: https://doi.org/10.1007/978-1-4419-7719-9_6.

[295] L. Szekeres, M. Payer, T. Wei, and D. Song, “SoK: Eternal War in Memory,” in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[296] M. Lindner, N. Fitinghoff, J. Eriksson, and P. Lindgren, “Verification of Safety Functions Implemented in Rust - a Symbolic Execution based approach,” in *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, vol. 1, 2019, pp. 432–439. DOI: 10.1109/INDIN41052.2019.8972014.

[297] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, “Safe Systems Programming in Rust,” *Communications of the ACM*, vol. 64, no. 4, pp. 144–52, Mar. 2021, ISSN: 0001-0782. DOI: 10.1145/3418295.

[298] A. N. Evans, B. Campbell, and M. L. Soffa, “Is Rust Used Safely by Software Developers?” In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 246–257. [Online]. Available: https://ieeexplore.ieee.org/document/9283950.

[299] L. Prechelt, “An empirical comparison of seven programming languages,” *Computer*, vol. 33, no. 10, pp. 23–29, 2000. DOI: 10.1109/2.876288.

[300] S. Raschka, J. Patterson, and C. Nolet, “Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence,” *Information*, vol. 11, no. 4, 2020, ISSN: 2078-2489. DOI: 10.3390/info11040193. [Online]. Available: https://www.mdpi.com/2078-2489/11/4/193.

[301] S. Cass, “The top programming languages: Our latest rankings put Python on top-again,” *IEEE Spectrum*, vol. 57, no. 8, pp. 22–22, 2020. DOI: 10.1109/MSPEC.2020.9150550.

[302] G. K. Kloss, “Automatic C library wrapping Ctypes from the trenches,” 2009.

[303] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, “Cython: The Best of Both Worlds,” *Computing in Science Engineering*, vol. 13, no. 2, pp. 31–39, 2011. DOI: 10.1109/MCSE.2010.118.

[304] K. Rakowski, *Learning Apache Thrift*. Packt Publishing Ltd, 2015.

[305] D. Abrahams and R. W. Grosse-Kunstleve, “Building hybrid systems with Boost. Python,” *CC Plus Plus Users Journal*, vol. 21, no. 7, pp. 29–36, 2003.

[306] M. Grinberg, *Flask Web Development: Developing Web Applications with Python*. O’Reilly Media, Inc., 2018.

[307] M. Giacobbe, C. Chaouch, M. Scarpa, and A. Puliafito, “An Implementation of InfluxDB for Monitoring and Analytics in Distributed IoT Environments,” in *Proceedings of the 8th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT’18), Vol.1*, M. S. Bouhlel and S. Rovetta, Eds., Cham: Springer International Publishing, 2020, pp. 155–162, ISBN: 978-3-030-21005-2.

[308] Beermann, Thomas, Alekseev, Aleksandr, Baberis, Dario, Crépé-Renaudin, Sabine, Elmsheuser, Johannes, Glushkov, Ivan, Svatos, Michal, Vartapetian, Armen, Vokac, Petr, and Wolters, Helmut, “Implementation of ATLAS Distributed Computing monitoring dashboards using InfluxDB and Grafana,” in *24th International Conference on Computing in High Energy and Nuclear Physics (CHEP 2019)*, vol. 245, Nov. 2020, p. 03 031. DOI: 10.1051/epjconf/202024503031.

[309] L. Junyan, X. Shiguo, and L. Yijie, “Application Research of Embedded Database SQLite,” in *2009 International Forum on Information Technology and Applications*, vol. 2, 2009, pp. 539–543. DOI: 10.1109/IFITA.2009.408.

[310] J. D. Drake and J. C. Worsley, *Practical PostgreSQL*. O’Reilly Media, Inc., 2002.

[311] D. Jaramillo, D. V. Nguyen, and R. Smart, “Leveraging microservices architecture by using Docker technology,” in *IEEE SoutheastCon 2016*, 2016, pp. 1–5. DOI: 10.1109/SECON.2016.7506647.

[312] M. Alam, J. Rufino, J. Ferreira, S. H. Ahmed, N. Shah, and Y. Chen, “Orchestration of Microservices for IoT Using Docker and Edge Computing,” *IEEE Communications Magazine*, vol. 56, no. 9, pp. 118–123, 2018. DOI: 10.1109/MCOM.2018.1701233.

[313] O. Ben-Kiki, C. Evans, and B. Ingerson, “Yaml ain’t markup language (yaml™) version 1.1,” *Working Draft 2008-05*, vol. 11, 2009.

[314] P. J. Eby, “Python Web Server Gateway Interface v1.0.1,” Python Software Foundation, Tech. Rep. PEP 3333, Sep. 2010.

[315] J. Shah and D. Dubaria, “Building Modern Clouds: Using Docker, Kubernetes Google Cloud Platform,” in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, 2019, pp. 0184–0189. DOI: 10.1109/CCWC.2019.8666479.

[316] M. W. Lucas, *TLS Mastery*. Tilted Windmill Press, 2020, ISBN: 978-1-64235-053-1.

[317] J. Adams, P. Elwell, A. Scheller, R. Getz, P. Rosenberger, J. Hughes, and J. Hickey, *Raspberry Pi Add-on Boards and HATs Specification*, Raspberry Pi Foundation, Cambridge, UK, 2014. [Online]. Available: https://github.com/raspberrypi/hats.

[318] M. Eremia, L. Toma, and M. Sanduleac, “The Smart City Concept in the 21st Century,” *Procedia Engineering*, vol. 181, pp. 12–19, 2017, 10th International Conference Interdisciplinarity in Engineering, INTER-ENG 2016, 6-7 October 2016, Tirgu Mures, Romania, ISSN: 1877-7058. DOI: https://doi.org/10.1016/j.proeng.2017.02.357. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877705817309402.

[319] A. J. Poulter, *The Secure Remote Update Protocol (SRUP) in Action*, Video, May 2021. [Online]. Available: https://youtu.be/F0_qlqh0Oiw.

[320] ——, *SRUP Timing Experiment*, University of Southampton, 2017. DOI: 10.5258/SOTON/D0486. [Online]. Available: https://doi.org/10.5258/SOTON/D0486.

[321] A. N. Kuznetsov, *Iproute2 routing commands and utilities*, https://git.kernel.org/, 2001. [Online]. Available: https://man7.org/linux/man-pages/man8/tc.8.html.

[322] T. Hombashi. (Aug. 2020). “tcconfig: A tc command wrapper.,” GitHub, [Online]. Available: https://github.com/thombashi/tcconfig.

[323] A. S. Khatouni, M. Trevisan, and D. Giordano, “Data-Driven Emulation of Mobile Access Networks,” in *2019 15th International Conference on Network and Service Management (CNSM)*, Oct. 2019, pp. 1–6. DOI: 10.23919/CNSM46954.2019.9012691.

[324] M. Trevisan. (2019). “Mobile Network Latency Emulator,” GitHub, [Online]. Available: https://github.com/marty90/mobile-latency-emulator.

[325] M2Catalyst LLC, *Network cell info app*, M2Catalyst LLC. [Online]. Available: https://m2catalyst.com/apps/network-cell-info.

[326] A. Sultan and M. Pope, *Digital cellular telecommunications system (phase 2+) (gsm); universal mobile telecommunications system (umts); network architecture*, European Telecommunications Standards Institute, 1999.

[327] P. Schramm, H. Andreasson, C. Edholm, N. Edvardsson, M. Hook, S. Javerbring, F. Muller, and J. Skold, “Radio interface performance of EDGE, a proposal for enhanced data rates in existing digital cellular systems,” in *VTC ’98. 48th IEEE Vehicular Technology Conference. Pathway to Global Wireless Revolution (Cat. No.98CH36151)*, vol. 2, 1998, 1064–1068 vol.2. DOI: 10.1109/VETEC.1998.686403.

[328] H. P. Naper and M. Pope, *Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); General Packet Radio Service (GPRS); Service description; Stage 2*, 1999.

[329] M. Ghaderi and R. Boutaba, “Data Service Performance Analysis in GPRS Systems,” in *2004 IEEE 15th International Symposium on Personal, Indoor and Mobile Radio Communications (IEEE Cat. No.04TH8754)*, vol. 1, Sep. 2004, 556–560 Vol.1. DOI: 10.1109/PIMRC.2004.1370932.

[330] P. Goyal and A. Goyal, “Comparative study of two most popular packet sniffing tools - Tcpdump and Wireshark,” in *2017 9th International Conference on Computational Intelligence and Communication Networks (CICN)*, 2017, pp. 77–81. DOI: 10.1109/CICN.2017.8319360.

[331] J. M. Perkel, “Why Jupyter is data scientists’ computational notebook of choice,” *Nature news*, vol. 563, no. 7732, pp. 145–146, Oct. 2018. DOI: 10.1038/d41586-018-07196-1.

[332] I. Stančin and A. Jović, “An overview and comparison of free Python libraries for data mining and big data analysis,” in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2019, pp. 977–982. DOI: 10.23919/MIPRO.2019.8757088.

[333] A. J. Poulter, *Dataset for An assessment of the Performance of the Secure Remote Update Performance in Simulated Real-World Conditions*, University of Southampton, May 2021. DOI: 10.5258/SOTON/D1817.

[334] A. Zourmand, A. L. Kun Hing, C. Wai Hung, and M. AbdulRehman, “Internet of things (iot) using lora technology,” in *2019 IEEE International Conference on Automatic Control and Intelligent Systems (I2CACIS)*, 2019, pp. 324–330. DOI: 10.1109/I2CACIS.2019.8825008.

[335] A. Lavric, A. I. Petrariu, and V. Popa, “SigFox Communication Protocol: The New Era of IoT?” In *2019 International Conference on Sensing and Instrumentation in IoT Era (ISSI)*, 2019, pp. 1–4. DOI: 10.1109/ISSI47111.2019.9043727.

[336] A. J. Poulter, *The Secure Remote Update Protocol - A Specification*, Unviersity of Southampton, Jul. 2017. DOI: 10.5258/SOTON/D0232.

[337] Department for Digital, Culture, Media & Sport, “Secure by Design: Improving the cyber security of consumer Internet of Things,” HM Government (UK), Tech. Rep., Mar. 2018. [Online]. Available: https://www.gov.uk/government/publications/secure-by-design-report.

[338] J. D’Abruzzo Pereira and M. Vieira, “On the Use of Open-Source C/C++ Static Analysis Tools in Large Projects,” in *2020 16th European Dependable Computing Conference (EDCC)*, 2020, pp. 97–102. DOI: 10.1109/EDCC51268.2020.00025.

[339] N. Nethercote and J. Seward, “Valgrind: A Program Supervision Framework,” *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 2, pp. 44–66, 2003, RV ’2003, Run-time Verification (Satellite Workshop of CAV ’03), ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(04)81042-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066104810429.

[340] S. Thénault, *Pylint – code analysis for Python*, comp. software, 2006. [Online]. Available: https://pylint.org/.

[341] G. Van Rossum, B. Warsaw, and N. Coghlan, “PEP 8: style guide for Python code,” Python.org, Tech. Rep., Jul. 2001. [Online]. Available: https://www.python.org/dev/peps/pep-0008/.

[342] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[343] A. Sen, “A quick introduction to the Google C++ Testing Framework,” *IBM DeveloperWorks*, vol. 20, pp. 1–10, 2010.

[344] J. Hunt, “PyTest Testing Framework,” in *Advanced Guide to Python 3 Programming*. Cham: Springer International Publishing, 2019, pp. 175–186, ISBN: 978-3-030-25943-3. DOI: 10.1007/978-3-030-25943-3_15. [Online]. Available: https://doi.org/10.1007/978-3-030-25943-3_15.

[345] P. Hamill, *Unit test frameworks: tools for high-quality software development*. O’Reilly Media, Inc., 2004, ISBN: 978-0596006891.

[346] Sutton, Michael and Greene, Adam and Amini, Pedram, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, Jun. 2007, ISBN: 978-0321446114.

[347] A. Shostack, *Threat modeling: Designing for security*. John Wiley & Sons, Feb. 2014, ISBN: 978-1118809990.

[348] P. Torr, “Demystifying the threat modeling process,” *IEEE Security Privacy*, vol. 3, no. 5, pp. 66–70, 2005. DOI: 10.1109/MSP.2005.119.

[349] J. M. Spivey, *The Z Notation: A Reference Manual*. Prentice Hall International, 1989.

[350] N. A. El-Araby, A. M. Wahba, and M. M. Taher, “Implementation of formally verified real time distributed systems: Simplified flight control system,” in *The 2011 International Conference on Computer Engineering Systems*, 2011, pp. 25–32. DOI: 10.1109/ICCES.2011.6141006.

[351] Z. Jiang, M. Pajic, S. Moarref, R. Alur, and R. Mangharam, “Modeling and Verification of a Dual Chamber Implantable Pacemaker,” in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Flanagan and B. König, Eds., Springer Berlin Heidelberg, 2012, pp. 188–203.