Secured Remote Update Protocol in IoT Data Exchange System

Bohdan Zhurakovskyi[1], Oleksandr Pliushch[2], Mikhail Polishchuk[1], Nataliia Korshun[3], and Sergiy Obushnyi[3]

[1]*National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute," 37 Peremogy ave., Kyiv, 03056, Ukraine*
[2]*Taras Shevchenko National University of Kyiv, 60 Volodymyrska str., Kyiv, 01601, Ukraine*
[3]*Borys Grinchenko Kyiv University, 18/2 Bulvarno-Kudriavska str., Kyiv, 04053, Ukraine*

Abstract

The article researches and develops a system for secure data exchange in IoT. The architecture of the secure remote update protocol and the types of messages supported by this protocol have been developed, and look at how the protocol can be used with the MQTT publish/subscribe architecture. Issues related to identity management for IoT devices are discussed, as well as an automated system identity management scheme to be used with the protocol. An experimental evaluation of the protocol and system and further analysis of the obtained data was carried out. All basic elements of the protocol are ready-made components. It was built on top of the very widely used MQTT protocol. It uses TLS to encrypt messages and uses RSA and SHA-256 to sign messages in the reference implementation. This use of well-known components lends some credibility to the protocol and makes it easier for users to understand.

Keywords

System for secure data exchange, secure remote update protocol, command and control system, message encryption, software library.

1. Introduction

IoT security is a technology segment focused on protecting connected devices and networks in the Internet of Things (IoT). IoT involves the addition of Internet connectivity to a system of interconnected computing devices, mechanical and digital machines, objects, animals, and/or people [1]. Each "thing" is given a unique identifier and the ability to automatically transfer data over the network. Allowing devices to connect to the Internet exposes them to many serious vulnerabilities if not properly secured [2].

Several high-profile incidents where a common IoT device was used to infiltrate and attack a larger network have drawn attention to the need for IoT security [3–5]. It is important to ensure the security of networks with IoT devices connected to them [6]. IoT security includes a wide range of techniques, strategies, protocols, and actions aimed at mitigating the increasing vulnerability of today's business in the IoT [7].

A command and control system (C2), although traditionally considered in a military context, simply means the process by which a hierarchically superior entity sets tasks for (command) or gives direct instructions (control) to a subordinate entity [8].

In the context of this study, C2 can be seen as a reference to any communication that occurs in a hierarchical structure where information is exchanged between a controlling entity [9] and a subordinate entity to ensure the management of the subordinate entity's activities by a higher controlling body.

There are two main types of architectures: a purely hierarchical approach, typically used in the context of military or civilian law enforcement personnel, and a peer-to-peer approach.

CPITS 2023: Workshop on Cybersecurity Providing in Information and Telecommunication Systems, February 28, 2023, Kyiv, Ukraine
EMAIL: zhurakovskiybyu@tk.kpi.ua (B. Zhurakovskyi); opliusch@yahoo.com (O. Pliushch); borchiv@ukr.net (M. Polishchuck); n.korshun@kubg.edu.ua (N. Korshun); s.obushnyi@kubg.edu.ua (S. Obushnyi)
ORCID: 0000-0003-3990-5205 (B. Zhurakovskyi); 0000-0001-5310-0660 (O. Pliushch); 0000-0003-0824-7895 (M. Polishchuck); 0000-0003-2908-970X (N. Korshun); 0000-0001-6936-955X (S. Obushnyi)

In the strictly hierarchical C2 communication model, all communication between nodes must go through a (common) senior commander.

In contrast to this approach, a purely peer-to-peer communication model allows direct communication between any nodes. An architecture that adopts this scheme will have a completely flat hierarchy where all nodes are equal and any communications flow directly between endpoints. More typically, in nominally peer-to-peer communications (such as a telephone system, for example), messages are routed or relayed through a server, exchange, or other intermediaries at one or more levels of hierarchical abstraction from the user [8]. This is still the case for most End-to-End Encryption (E2EE) messaging services such as Signal 3.

This paper primarily considers a use case scenario commonly encountered in IoT—where a certain class (or family of classes) of devices is deployed and C2 functionality provided by a centralized service [10] is used. This service can support one or more users, each of whom will have access to one or more devices.

It is also compatible with the new trend of using IoT devices and integrating with existing cloud services such as those provided by Google or Amazon. This approach enables easy integration with home automation hub devices such as AmazonEcho (and Alexa voice assistant) and GoogleHome [11].

Given the requirements for IoT and IoT systems in a military context, there is a clear requirement for a C2 system to integrate them into existing military decision-making processes and structures.

The infrastructure requirements underlying the network connectivity for deploying IoT devices can be met by deployed military systems or local theater systems (or a combination of both). However, depending on the scale of any such deployment of IoT devices, the infrastructure may not be sufficient to meet the connectivity requirements to support a real-time system [12]. The use of mesh networks consisting of device-to-device communication and more sophisticated network management can solve some of these problems.

Finally, it should be noted that the technical challenges of messaging in C2 systems differ from the organizational challenges of decision-making, information management, and situational awareness that future IoT systems will also face.

2. Statement of the Research Problem

2.1. System Development

As an example, consider the case where there is a deployed IoT system consisting of a fixed number of sensor devices, and if the devices belong to a C2 (Command and Control) control system, they all work as part of a smart city. In the event of an emergency, these devices can be supplemented by pairing them with a second set of mobile devices operated by local fire and rescue services. The protocol supports the dynamic addition of devices to the C2 network, as well as commands that require devices to transfer their registration to another C2 server if the corresponding systems work on the same network [13].

For this paper, we consider the use of a network of fixed sensors installed to measure air quality and provide real-time alerts to citizens if air quality deteriorates below a certain threshold.

In the event of a major fire or other disasters, it may be desirable to supplement this system with several additional mobile sensors, such as those that can be deployed by the local fire department. As part of the development, this could easily be achieved if the fire brigade sensors are commanded to join the city's C2 network. This combined system C2 is shown in Fig. 1.

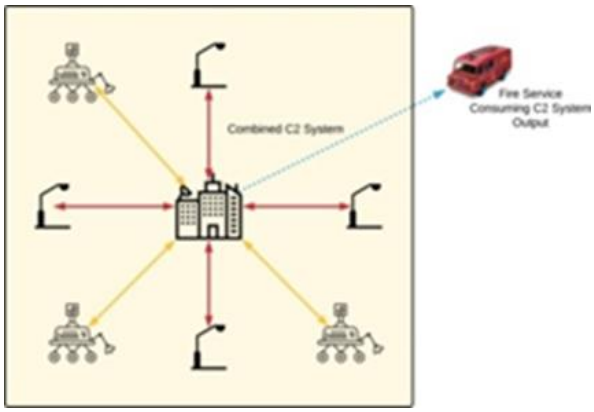

Figure 1: Combined system C2

More generally, for this to be possible if all of the following three conditions are true:

1. Systems must use (or be compatible and able to use) the same protocol universe.
2. The owner of the sensors to be connected to the existing C2 system agrees to (temporarily) transfer ownership of the sensors to the operator of the fixed system.
3. The owner of the fixed system accepts the new sensors and assumes temporary responsibility for them.

Conditions 2 and 3 are particularly important. Since the devices in question will become full members of the new C2 network, both parties must be prepared to join the systems, which involves a certain degree of trust. For example, the party hosting the C2 network must be confident that the devices that join are not compromised by malware, and the party providing the devices must be confident that the other party will make every effort to ensure that they are not and that they will be returned to their control after the operation is completed [14].

However, there are scenarios where such unification cannot be achieved because one or more of the preconditions are not met and cannot be satisfied (for example, when one party does not wish to assign its devices to the control of the C2 network operator—or if the devices in question necessarily use different server systems).

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

2.2. Web System C2

As part of this study, an implementation of the C2 web system was created. This was implemented in Python because the protocol library [15] could be used. The C2 system creates a Server class with ruSRUP. In this example, the Flask5 library was used to provide a framework for web development, although any other framework can be substituted very easily [16]. Flask was adopted because it provides a very lightweight web development framework, supporting Jinja6 templates for generating dynamic pages. Any real-life C2 system is inherently application-specific, so this example implementation is only included as reference material for developers implementing their custom systems. Architectural scheme Fig. 2.

Setting up a C2 server requires some additional steps compared to 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 credentials simply by visiting a key exchange URL), the registration process for servers requires the creation of a server token file. This, along with the server ID, must be specified in the manually generated configuration file. The C2 server reference implementation also supports human- and machine-driven joins. It allows you to use color or monochrome icons, word lists, and hexadecimal notation. It also includes a simple example of visualizing data from a device by plotting a graph, although in a real (production) system it would be highly desirable to use scalable, off-the-shelf time series data storage and visualization tools such as InfluxDb and Grafana.

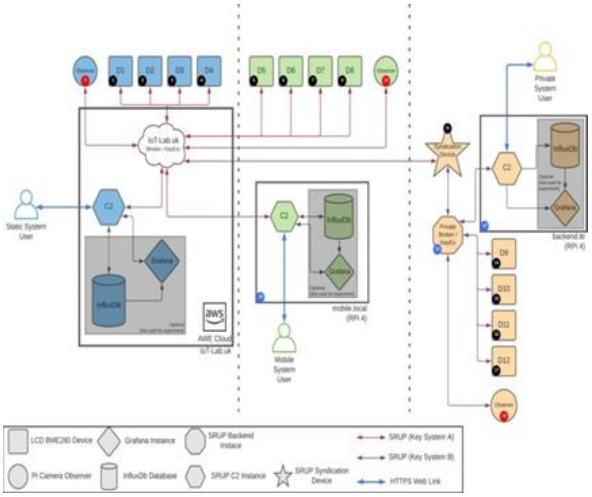

Figure 2: Architectural scheme

It should be noted that the implemented C2 system does not use any type of client-side user authentication. However, implementing user authentication for web applications is a solved problem, and there are several ready-made OSS solutions compatible with Python implementations (such as Flask-Login7) that can be easily added for any real-world use.

2.2.1. Backend Services

A reference implementation was also created for the systems server services. It consists of two main parts: a key exchange server; and supporting infrastructure for hosting and brokering MQTT messages.

The key exchange server was implemented as a relatively simple Python program. It implements a REST API endpoint for all steps of the device registration process and again uses the Flask library to provide a web services framework [17]. All data (such as device IDs and keys) is stored locally using an SQLite database, however for a larger full production system, this can easily be replaced with any other SQL database system (eg PostgreSQL) [18].

Table 1 shows the implemented REST endpoints.

Although the key exchange server is relatively simple (especially when running locally), the configuration required to host it securely on a remote web server [19] (including providing a TLS certificate for the hosting domain) and the configuration required by the MQTT broker represents additional steps. Using Docker and the Docker-compose orchestration framework, it is easy (and reproducible) to specify a persistent configuration for these components.

Table 1
REST endpoints

End-Point	Type	Service
../register/ status	GET	Returns the status of the KeyEx server
../register/ register	POST	Performs device ID registration
../register/ validate	POST	Performs mutual verification 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 the given device ID
../register/ get_type	GET	Returns the device type for the device ID
../register/ C_check	GET	The C2 server is registered with the KeyEx service
../register/ C	POST	Registers the C2 server in the system

The basic orchestration configuration is specified using the YAML (YAML Ain't Markup Language) format and is contained in the docker-compose.yml file [20, 21]. This defines four microservices, each implemented as a docker container. They are described in Table 2.

Of these, only the KeyEx service is an individual container. The other three all use existing container images and augment them with custom configuration settings specified in the docker build file. They are presented in containers through the file system using Docker Volumes.

The Dockerfile that defines the build process for a KeyEx image simply takes the latest Python 3 container image and installs the Python library dependencies for KeyEx. These include Flask, a Cryptography 8 library, and Green Unicorn9, which implements Python.

Web Server Gateway Interface (WSGI). The last step is to define a startup script for the KeyEx service [21].

Table 2
Docker containers

Service Name	Container Image	Service description
web	nginx: latest	Implements nginx as a reverse proxy web server
keyex	Bespoke	Implements the key service
broker	eclipse-mosquitto: latest	Implements Eclipse Mosquitto MQTT broker
certbot	certbot/certbot	Implements the Let's Encrypt certification robot to generate a TLS certificate

Thus, the entire backend can be distributed as a series of small source code files that can be assembled using Docker and Docker-Compose to deploy the complete SRUP universe on any server with minimal developer intervention. Although Docker-Compose was adopted as a reference implementation, alternative orchestration layers such as Kubernetes can be substituted, especially for running large-scale deployments under high or variable loads [21].

An architectural diagram showing how all the components fit together is shown in Fig. 3.

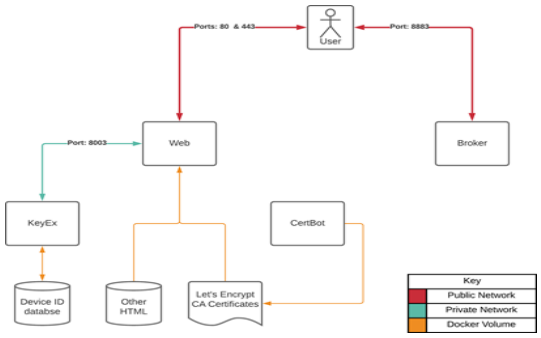

Figure 3: Architectural diagram of components

2.2.2. Confirmation of Identity

Using dynamic device identification requires solving the problem of mapping a particular physical device to whatever logical device identification it has at any point in time. In particular, this is required when this device is connected to the network of the C2 system. If a device's identity can be established at the time it joins, it can be maintained throughout its participation in the C2 system network.

The simplest form of connection operation is a simple or unmoderated connection. It does not

attempt to verify the ID of the connecting device and only accepts the request based on the credentials provided (such as the device type) during the initial device registration.

This type of association is only suitable for situations where either there is no risk of a malicious device being added (for example, on a closed network within a secure perimeter) or where the overall system is not adversely affected by erroneous or malicious data being sent from the device. Given the MQTT addressing model [22] and the use of topic access control, there is not much danger that a malicious device will be able to intercept or corrupt messages to and from other devices (assuming that the MQTT broker implementation is error-free and resistant to attempts to apply such methods such as buffer overflow [23].

2.3. Development of a Secure Remote Update Protocol

2.3.1. Protocol Library Architecture

To combine the best elements of a binary implementation with the ease of use of a scripting language, a hybrid approach was taken to implement the SRUP software library. For this study, the binary code was written in C++, and Python was selected as the scripting language.

The design concept was to create a basic implementation to generate and process a stream of bytes to be used as the payload of an MQTT message [23]. This code was implemented using C++ and a binary library (libSRUP_Lib) was formed [24–26]. Using this approach ensured the efficient inclusion of existing OSS binaries for cryptographic functions (such as RSA signatures) by including these libraries (eg libcrypto—part of OpenSSL2) in the build process.

A Python wrapper (ruSRUPLib) [27] was implemented to allow direct use of the Python binary library implementation of the protocol and was then wrapped in a pure Python library (ruSRUP) consisting of classes designed to be called directly from user application code. This Python wrapper was designed to implement as much of SRUP's common functionality as possible (such as providing valid sequence IDs), and thus significantly reduced the implementation for building a program using it.

An architectural diagram illustrating this approach can be seen in Fig. 4.

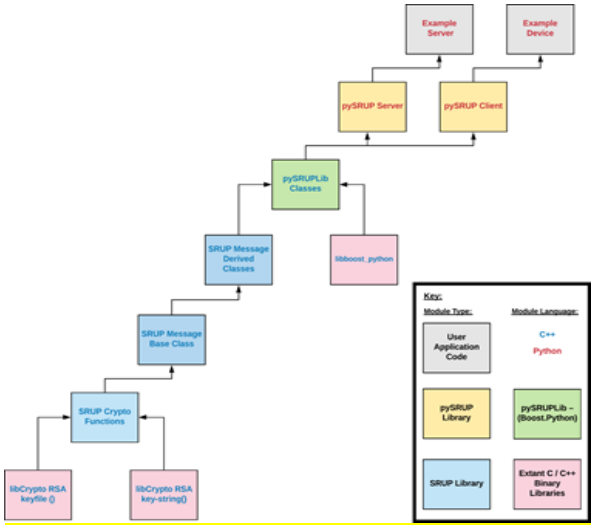

Figure 4: System architecture

In particular, the following elements were implemented as part of this work:

1. C++ library (libSRUP_Lib).
2. Python binary library (ruSRUPLib).
3. Pure Python shell class (ruSRUP).
4. Key exchange server.
5. C2 web system.
6. Container backend.
7. Initial key generation tool.

2.3.2. Bootstrapping the Protocol and Key Generation Tool

The final piece needed to ensure the protocol is created from scratch is a means to generate the set of required certificates and keys. A total of eight of the following files are required for a system running on a network connected to the Internet:

- CA certificate of the broker.
- A pair of public and private keys of the C2 server (used to identify the C2 server).
- C2 server token file.
- Pair of public and private keys of the key exchange server (used to authenticate the KeyEx server).
- C2 broker access key and certificate (used for C2 server access to the MQTT broker).

A system running on a private network (or a network not connected to the Internet) also requires two additional files: the private root key of web CA; the private web CA root certificate.

The process of generating these files, especially CA root files, is quite complex. So a key generation tool (also written in Python) was created to be shared with other backend elements.

This tool is designed to generate the initial bootstrap configuration for a new base system installation and to generate the necessary credentials (including a server token file) for any C2 servers to be used on that system.

Fig. 5 shows the system at work and the relationships between the various files created.

Using Docker, the container approach creates the backend as a component that the user can easily deploy without having to implement the complexities of securing the broker and managing the key exchange itself [28].

Thus, this approach allows the deployment of the SRUP universe to be considered as a commodity service to individuals or organizations that have a particular requirement for the privacy or security of their IoT systems [29], they can provide the necessary server systems for their IoT devices within their own controlled network infrastructure or hosting environments, and thus be able to provide guarantees about the availability and availability of TLS keys used to encrypt MQTT traffic [22].

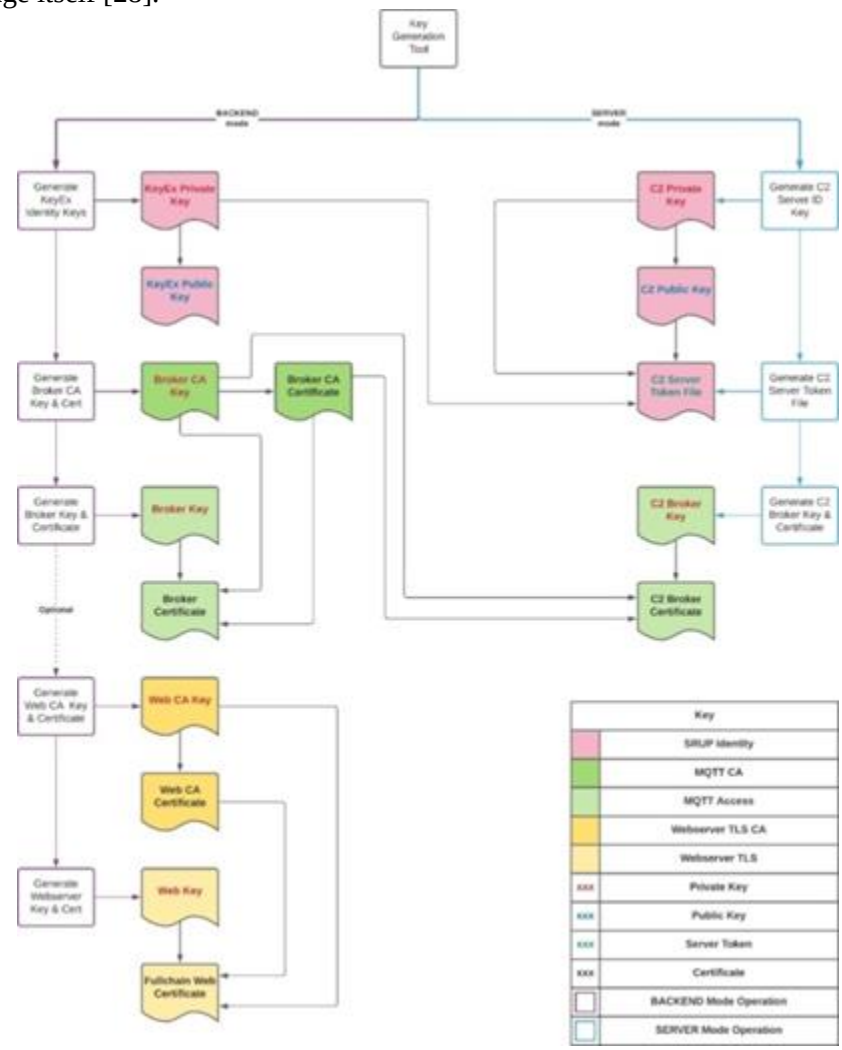

Figure 5: System operation scheme

3. Experimental Assessment of System Performance

Experiments are performed and described in this section to compare the relative processing time, message size, and power consumption of a system using the protocol with an identical device that uses an insecure approach and uses regular MQTT messages directly.

3.1. Execution Time Analysis

An initial evaluation of the protocol's performance was performed by determining the execution time of the cryptographic functions used to support the protocol. A test case was evaluated by measuring the time required to sign and verify a representative update initiation message. This process was run five times on a

Raspberry Pi 3 B with the average time calculated. According to the standard implementation [30], the signature functions used SHA-256 to generate a hash of the message, and then an RSA signature was calculated for this hash value. This signing process took an average of 56.68 ms, and the average signature verification time was only 9.910 ms.

3.2. Hardware

The experimental system consisted of five IoT devices, each built on a Raspberry Pi 3B+ single-board computer equipped with a special board with two LED status indicators and a button for user interaction.

The devices were connected via Ethernet to a Raspberry Pi 3B+, which acted as the C2 server. The C2 server ran custom software that randomly selected one of five devices, sent a message to that device asking it to change its LED state, and then waited for a random interval before returning. The program continued until each device received 250 messages. This workflow is shown in Fig. 6. Two separate implementations of the C2 system software were written. One uses protocol action messages over a TLS-encrypted MQTT connection, and the other uses a plain text MQTT message.

The previous assumption was that a significant proportion of any additional delay would be due to the time spent processing the cryptographic algorithms used to sign the messages. To assess the extent to which the performance of the protocol depends on hardware speed, an additional device was created using a faster Raspberry Pi 4 single-board computer, allowing performance to be compared with other devices. The Pi 4 was identical to the devices described above except for the CPU and RAM configuration changes between the Pi 3 and Pi 4.

3.3. Experimental Measurement

When using the protocol in the system, an increase in the time needed to process messages was expected, as well as an increase in the power consumption of system devices (due to additional processing requirements for running message signature algorithms). The total size of the data sent had to be increased as well (due to the additional fields used by SRUP to ensure message security).

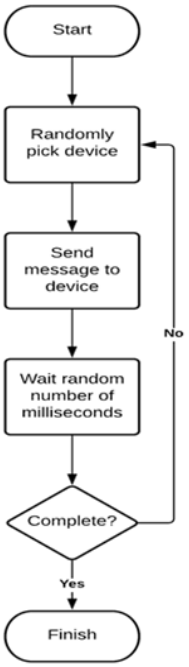

Figure 6: Block diagram of the program

When testing the system, the following measurements were made:

1. Time. The actual performance measurement was evaluated by analyzing the log files generated by the devices and the server for each experimental run.
2. Power. The average power consumption of one of the devices during MQTT and SRUP conditions was evaluated. The measurement was made using a USB power meter.
3. Message size. Network traffic was captured using Wireshark and examined to determine the size of raw MQTT messages and the MQTT implementation of the system protocol.

3.4. Analysis of System Testing Results

All log file analysis was done using Python and Jupyter. The panda's library was used to map and analyze the data.

This analysis included:

1. Mapping the device ID to the logical device number.
2. Download log files from the C2 server for each experiment.
3. Removing unused columns from the resulting data frame.
4. Downloading all log files from each device for each experiment and concatenating

them into a single Python object (a list of data frame dictionaries).

5. Create a new data frame for each line in log frame C2, recording the device number, operation type (enabled or disabled), and the timestamp when the command was sent.
6. For each row in the data frame created in step 5, extract the timestamp when that command was received by the device.
7. Calculation of the delay time between sending and receiving, in milliseconds.

Each experiment produced a graph that was used to verify the data acquisition process (Figs. 7 and 8). As expected, experiments using a delay distribution had a significantly larger standard deviation. A similar analysis process was also performed for experimental runs using MQTT.

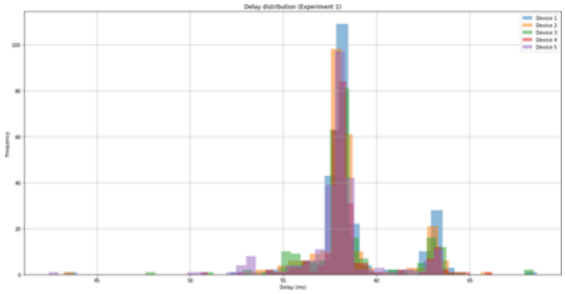

Figure 7: A graph showing the distribution of latency associated with SRUP message propagation and processing time

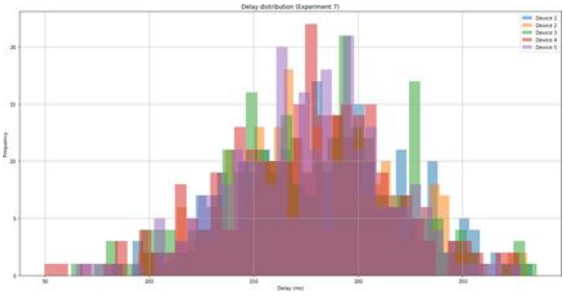

Figure 8: A plot showing the distribution of latency associated with SRUP message propagation and processing time for an experiment.

A graph showing the average latency for each experiment is shown in Fig. 9, and a graph showing the combined averages for each device for a given experiment (for both the protocol and MQTT cases) is shown in Fig. 10. General the average processing overhead for the protocol compared to MQTT across all network conditions showed an additional 51.60 ms. This compares to a worst case of 56.13 ms excluding the effects of network latency. Although the added latency is much smaller (58.44 ms vs. 2.31 ms) in Ethernet LAN compared to a fully unprotected system, compared to a more representative scenario for a deployed IoT system (average 4G capacity): the overhead is only 53.55% of the delay MQTT (147.7 ms vs 96.17 ms = 51.53 ms).

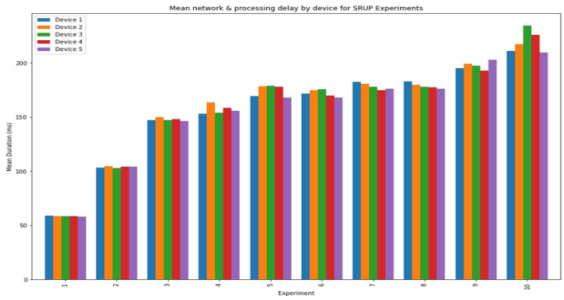

Figure 9: Graph showing the average message network and processing latency by device for each of the experiments

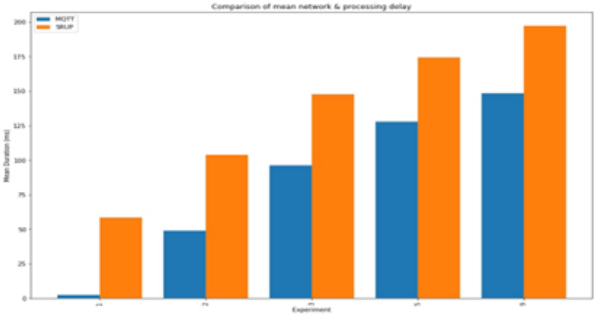

Figure 10: A graph showing the overall average network and processing latency for MQTT messages

Even in the worst case, the processing overhead means that only if the message frequency exceeds 17.82 Hz will the additional processing time be greater than the natural period of the message (equation 1).

$$\frac{1}{56.13\,\text{ms}} = 17.82\,\text{Hz}$$

Since a typical real-world IoT device can be expected to have an average message-to-message time of minutes, the additional processing overhead on the order of tens of milliseconds is a very small additional price for the very significant security benefits provided by the protocol. However, this result shows that the protocol in its current form may not be suitable for very time-critical applications when running on lower-spec hardware. The analysis also shows that the system's protocol is robust even under extremely poor network conditions. Even in the worst case, all messages were correctly received within 4026 ms (Fig. 11) due to the robust nature of the underlying MQTT protocol.

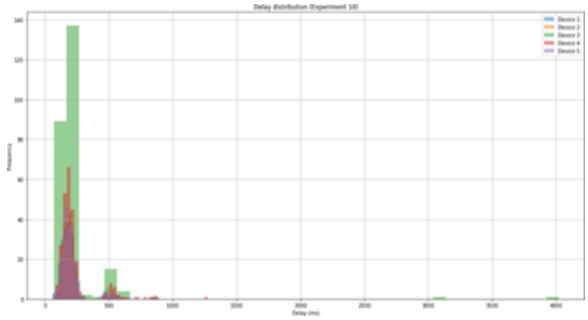

Figure 11: A graph showing the distribution of latency associated with message propagation and processing time

4. Conclusions

In this work, a system for secure data exchange in IoT has been researched and developed. The architecture of the secure remote update protocol and the types of messages supported by this protocol have been developed, and looked at as to how the protocol can be used with the MQTT publish/subscribe architecture. Issues related to identity management for IoT devices are discussed, as well as an automated system identity management scheme to be used with the protocol.

An experimental evaluation of the protocol and system and further analysis of the obtained data was carried out. The protocol (compared to a plain, insecure MQTT implementation) was shown to have an overall message processing latency of 51.60 ms to 42.92 ms when running on Raspberry Pi hardware, and an increased power consumption of 727.6 mW during message processing.

The Secure Remote Update Protocol provides an efficient design and implementation of secure C2 messaging for IoT applications. All basic elements of the protocol are ready-made components. It was built on top of the very widely used MQTT protocol (which itself runs over TCP). It uses TLS (including AES) to encrypt messages and (in the reference implementation) uses RSA and SHA-256 to sign messages. This use of well-known components lends some credibility to the protocol and makes it easier for users to understand.

Experimental evaluation of the system and protocol has shown that the system performs successfully even under extremely poor network conditions (far beyond those typically seen in the vast majority of real-world deployments). Experiments have shown that despite the performance overhead associated with using the protocol, it is a relatively small factor for most realistic use cases.

5. References

[1] Cybersecurity: Selected Cyberattacks, 2012–2021. Washington: Congressional Research Service, 2021. URL: https://crsreports.congress.gov/product/pdf/R/R46974

[2] B. Zhurakovskyi, et al., Coding for Information Systems Security and Viability, CEUR Workshop Proceedings, 2859 (2021) 71–84.

[3] A. Carlsson, et al., Sustainability Research of the Secure Wireless Communication System with Channel Reservation, in: 2020 IEEE 15th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering (2020). doi:10.1109/tcset49122.2020.235583

[4] I. Kuzminykh, et al., Investigation of the IoT device lifetime with secure data transmission, Internet of Things, Smart Spaces, and Next Generation Networks and Systems (2019) 16–27. doi: 10.1007/978-3-030-30859-9_2

[5] F. Kipchuk, et al., Investigation of Availability of Wireless Access Points based on Embedded Systems, in: 2019 IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (2019). doi: 10.1109/picst47496.2019.9061551

[6] What is a Cyber Attack? URL: https://www.ibm.com/topics/cyber-attack

[7] M. Moshenchenko, et al. Optimization Algorithms of Smart City Wireless Sensor Network Control, CEUR Workshop Proceedings, Cybersecur. Providing in Inf. and Telecommunication Systems II, 3188 (2021) 32–42.

[8] Internet of Things for Command and Control. URL:https://www.wonderbit.com/en/our-work/ncia-iot-for-c2

[9] IoT protocols and connectivity.URL: https://azure.microsoft.com/en-us/solutions/iot/iot-technology-protocols

[10] Constrained Application Protocol. URL:https://en.wikipedia.org/wiki/Constrained_Application_Protocol

[11] What is IPsec?, How IPsec VPNs work. URL: https://www.cloudflare.com/learning/network-layer/what-is-ipsec
[12] V. Druzhynin, et al., Features of Processing Signals from Stationary Radiation Sources in Multi-Position Radio Monitoring Systems, CEUR Workshop Proceedings, 2746 (2020) 46–65.
[13] B. Zhurakovskyi, et al., Calculation of Quality Indicators of the Future Multiservice Network, Future Intent-Based Networking, 831 (2022) 197–209. doi:10.1007/978-3-030-92435-5_11
[14] S. Obushnyi, et al., Autonomy of Economic Agents in Peer-to-Peer Systems, CEUR Workshop Proceedings, 3288 (2022) 125–133.
[15] What Is an X.509 Certificate? URL: https://www.ssl.com/faqs/what-is-an-x-509-certificate.
[16] Welcome to Flask—Flask Documentation. URL: https://flask.palletsprojects.com/en
[17] M. Straten, The Google Cloud powers your Philips Hue Lightbulbs, GDG DevFest Ukraine, 2018.
[18] O. Shevchenko, et. al, Methods of the Objects Identification and Recognition Research in the Networks with the IoT Concept Support, Workshop on Cybersecurity Providing in Inf. Telecommun. Syst. 2923 (2021) 277–282.
[19] B. Zhurakovskyi, et al., Modifications of the Correlation Method of Face Detection in Biometric Identification Systems, CEUR Workshop Proceedings, 3288 (2022) 55–63.
[20] YAML Syntax—Ansible Documentation. URL:https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax
[21] Docker Overview. URL: https://docs.docker.com/get-started/overview/
[22] MQTT: The Standard for IoT Messaging. URL: https://mqtt.org
[23] What Is Zigbee Protocol Wireless Mesh Networking? URL: https://www.digi.com/solutions/by-technology/zigbee-wireless-standard
[24] Radio-frequency Identification: URL: https://en.wikipedia.org/wiki/Radio-frequency_identification
[25] B. Zhurakovskyi, et al., Modifications of the Correlation Method of Face Detection in Biometric Identification Systems, CEUR Workshop Proceedings, 3288 (2022) 55–63.
[26] C++High-level Programming Language URL :https://uk.wikipedia.org/wiki/C%2B%2B
[27] Python (programming language). URL: https://en.wikipedia.org/wiki/Python_(programming_language)
[28] Y Hardware Attached on Top. URL: https://www.raspberrypi.com/news/introducing-raspberry-pi-hats
[29] N. Fedorova, et al., Software System for Processing and Visualization of Big Data Arrays, ICCSEEA 2022: Advances in Computer Science for Engineering and Education, 134 (2022) 324–336. doi:10.1007/978-3-031-04812-8_28
[30] B. Zhurakovskyi, et al., Modifications of the Correlation Method of Face Detection in Biometric Identification Systems, CEUR Workshop Proceedings, 3288 (2022) 55–63.