Contributions to Securing Software Updates in IoT

Nikbakht Bideh, Pegah

2022

Document Version:
Publisher's PDF, also known as Version of record

Contributions to Securing Software Updates in IoT

Pegah Nikbakht Bideh

LUND
UNIVERSITY

ISBN 978-91-8039-429-1 (printed)
ISBN 978-91-8039-430-7 (electronic)
Series of licentiate and doctoral theses
No. 152
ISSN 1654-790X

Pegah Nikbakht Bideh
Department of Electrical and Information Technology
Lund University
Box 118
SE-221 00 Lund
Sweden

Typeset using LaTeX.
Printed in Sweden by Tryckeriet i E-huset, Lund, 2022.

Dedicated to
Mahsa Amini
and other women who suffered in my country

Abstract

The Internet of Things (IoT) is a large network of connected devices. In IoT, devices can communicate with each other or back-end systems to transfer data or perform assigned tasks. Communication protocols used in IoT depend on target applications but usually require low bandwidth. On the other hand, IoT devices are constrained, having limited resources, including memory, power, and computational resources. Considering these limitations in IoT environments, it is difficult to implement best security practices. Consequently, network attacks can threaten devices or the data they transfer. Thus it is crucial to react quickly to emerging vulnerabilities. These vulnerabilities should be mitigated by firmware updates or other necessary updates securely. Since IoT devices usually connect to the network wirelessly, such updates can be performed Over-The-Air (OTA). This dissertation presents contributions to enable secure OTA software updates in IoT.

In order to perform secure updates, vulnerabilities must first be identified and assessed. In this dissertation, first, we present our contribution to designing a maturity model for vulnerability handling. Next, we analyze and compare common communication protocols and security practices regarding energy consumption. Finally, we describe our designed lightweight protocol for OTA updates targeting constrained IoT devices.

IoT devices and back-end systems often use incompatible protocols that are unable to interoperate securely. This dissertation also includes our contribution to designing a secure protocol translator for IoT. This translation is performed inside a Trusted Execution Environment (TEE) with TLS interception.

This dissertation also contains our contribution to key management and key distribution in IoT networks. In performing secure software updates, the IoT devices can be grouped since the updates target a large number of devices. Thus, prior to deploying updates, a group key needs to be established among group members. In this dissertation, we present our designed secure group key establishment scheme. Symmetric key cryptography can help to save IoT device resources at the cost of increased key management complexity. This trade-off can be improved by integrating IoT networks with cloud computing and Software Defined Networking (SDN). In this dissertation, we use SDN in cloud networks to provision symmetric keys efficiently and securely. These pieces together help software developers and maintainers identify vulnerabilities, provision secret keys, and perform lightweight secure OTA updates. Furthermore, they help devices and systems with incompatible protocols to be able to interoperate.

Acknowledgements

During my Ph.D., I was lucky to work with different supervisors. First, I would like to thank my first main supervisor, Martin Hell. Martin's guidance was really helpful to me, especially during the early years of my Ph.D. His advice helped me not only in research but also in teaching. I was always welcomed to Martin's office whenever I had questions. Martin was always quick in replying to emails or messages on Slack. This was unfortunate that Martin had to leave university in the last year of my Ph.D. However, I wish the best of luck to his company. I would like to thank my second main supervisor Christian Gehrmann, who became my main supervisor when Martin left the university. I thank Christian for all of his help during the last year of my Ph.D. In the last year, there was too much administrative stuff to do, and thanks to Christian to organize everything in the best way.

I would like to thank Nicolae Paladi, my assistant supervisor. I'm really thankful to get to know him during my Ph.D. Nicolae helped me a lot during my Ph.D.; his comments and guidance always gave me a clue on how to proceed. Nicolae was not only a supervisor but also a true friend that I could always talk. This world truly needs more people like you, Nicolae. It was also unfortunate that Nicolae had to leave the university in the last year of my Ph.D. However, I wish the best of luck to his startup company CanaryBit. Nicolae helped me not only in research but also in finding my career path after my Ph.D. I'm really grateful that I'm going to work with such great colleagues such as Nicolae and Stefano in CanaryBit after my Ph.D.

I would like to thank my other assistant supervisor Paul Stankovski Wagner. I didn't have the chance to collaborate on research with Paul; however, his guidance was really helpful to me in teaching. We always had great discussions during lunch. He followed the news even about my country Iran, and he sometimes knew some news about Iran before me, which always surprised me. Paul is such a nice, friendly, and caring supervisor; words can not really describe you, Paul; thank you for being part of my Ph.D. journey.

During these five years, I get to know different people in the department. First, I would like to thank the WASP crew, Alexander and Joakim. We started the WASP Ph.D. journey together, and I had the great companionship of Alexander and Joakim during all WASP courses, travels, and conferences. This journey would have been more difficult without you guys. Alexander and I also shared offices; I couldn't ask for a better officemate than him. I would like to thank Linus, we had the chance to collaborate in research and computer security labs together, and it was a great experience to work with him. Linus really helped me in the labs when he was here and even after his graduation. Sorry, Linus, if I asked too many questions. I would like to thank Jing; when she came, our group became more gender-balanced, and I didn't feel alone after that. I am grateful to Martin Gunnarsson; we always had great lunch discussions and other IoT-related discussions;

however, we didn't have the chance to collaborate. I also thank Denis and Hui for helping me in the computer security labs. I wish to thank the rest of the people in our research lab; it was a great pleasure to get to know all of you. I want to thank the administrative and technical staff at the department for helping me during my Ph.D.

I wish to thank the WASP organization for providing us with a valuable opportunity for research. A special thanks to Natalie Pintar and Anna Björnemo for their help in all the administrative stuff. They were always nice and helpful, even when we missed some registration deadlines.

My greatest thanks go to my husband, Sajad; without his help, I wouldn't be where I am today. I was lucky that Sajad also works in an IoT-related company, and I could always use his advice in my research. His unconditional love and support always lighten the way for me. There were difficult moments during these five years, and this was Sajad who always took my hand and gave me the strength to continue. Sajad! Thank you for being part of my life. I want to thank my mom and dad in Iran for their love and support. They were always there whenever I needed them.

مادر و پدر عزیزم دوستتون دارم و برای همه ی زحمت هایی که برای من کشیدین ازتون تشکر و قدردانی میکنم.

Pegah
Lund, December 2022

Contribution Statement

The following papers are included in this dissertation:

Paper I Pegah Nikbakht Bideh, Martin Höst, and Martin Hell. "HAVOSS: A Maturity Model for Handling Vulnerabilities in Third Party OSS Components". In *the 19th International Conference on Product-Focused Software Process Improvement, PROFES 2018, Wolfsburg, Germany*, pp. 81-97. Springer, Cham.

Paper II Pegah Nikbakht Bideh, Jonathan Sönnerup, and Martin Hell. "Energy Consumption for Securing Lightweight IoT Protocols". In *proceedings of the 10th International Conference on the Internet of Things, IoT 2020, Malmö, Sweden*, pp. 1-8, ACM.

Paper III Pegah Nikbakht Bideh and Christian Gehrmann. "RoSym: Robust Symmetric Key Based IoT Software Upgrade Over-the-Air". In *the 3rd Workshop on CPS&IOT Security and Privacy (CPSIOTSEC 2022), Los Angeles, U.S.A*.

Paper IV Pegah Nikbakht Bideh and Nicolae Paladi. "Chuchotage: In-line Software Network Protocol Translator for (D)TLS ". In *the 24th International Conference on Information and Communications Security, ICICS 2022, Kent, Canterbury, UK*, pp. 589-607. Springer, Cham.

Paper V Pegah Nikbakht Bideh. "LMGROUP: A Lightweight Multicast Group Key Management for IoT Networks". In *the 17th International Conference on Information Security Practice and Experience (ISPEC 2022), Taipei, Taiwan*.

Paper VI Nicolae Paladi, Tiloca, Marco, Pegah Nikbakht Bideh, and Martin Hell. "Flowrider: Fast On-Demand Key Provisioning for Cloud Networks". In *the 17th EAI International Conference on Security and Privacy in Communication Networks, EAI SecureComm 2021, Canterbury, Great Britain(online)*, pp. 207-228. Springer, Cham.

The table below summarizes the responsibilities Pegah Nikbakht Bideh had in each paper:

Paper	*Writing*	*Concepts*	*Implementation*	*Evaluation*
I	✓	✓	-	✓
II	**✔**	**✔**	✓	**✔**
III	**✔**	✓	**✔**	**✔**
IV	✓	✓	**✔**	**✔**
V	**✔**	**✔**	**✔**	**✔**
VI	✓			✓

Bold check marks indicate roles where Pegah Nikbakht Bideh took primary responsibility for the given role. The individual contributions of Pegah are described in more detail below.

In paper I, Pegah was involved in the writing, concepts, and evaluation of the paper. The paper did not include an implementation.

In paper II, Pegah had mainly the responsibility for writing and evaluating the paper. She contributed to the concepts and implementation of the paper.

In paper III, Pegah was responsible for the writing and concepts of the paper, and she solely implemented and evaluated the solution.

In paper IV, together with other authors, Pegah contributed to the writing and concepts of the paper. Her main responsibility was to implement and evaluate the solution.

In paper V, Pegah was the only author of the paper, and she was solely responsible for writing, concepts, implementation, and evaluation of the work.

In paper VI, Pegah's main responsibility was to verify the solution formally, and she contributed solely to writing that part and evaluating the security properties of the solution. She also contributed to the background part. The other authors contributed the most to concepts and implementation, and Pegah had a minor contribution in those parts.

A further description of the papers' contributions *to the research field* is presented in Section 3.1.

Other Contributions

The following publications have also been published during Pegah's Ph.D. studies but are not included in this dissertation.

- Nilsson, Alexander, Pegah Nikbakht Bideh, and Joakim Brorsson. "A survey of published attacks on Intel SGX." arXiv preprint arXiv:2006.13598 (2020).
- Nikbakht Bideh, Pegah, Nicolae Paladi, and Martin Hell. "Software-defined networking for emergency traffic management in smart cities." In *Vehicular Ad-hoc networks for smart cities*, pp. 59-70. Springer, Singapore, 2020.
- Paladi, Nicolae, Marco Tiloca, Pegah Nikbakht Bideh, and Martin Hell. "On-demand key distribution for cloud networks." In *2021 24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pp. 80-82. IEEE, 2021.
- Karlsson, Linus, Pegah Nikbakht Bideh, and Martin Hell. "A recommender system for user-specific vulnerability scoring." In *International Conference on Risks and Security of Internet and Systems*, pp. 355-364. Springer, Cham, 2019.
- Brorsson, Joakim, Pegah Nikbakht Bideh, Alexander Nilsson, and Martin Hell. "On the Suitability of Using SGX for Secure Key Storage in the Cloud." In *International Conference on Trust and Privacy in Digital Business*, pp. 32-47. Springer, Cham, 2020.

Contents

Introduction

Internet of Things (IoT) is an emerging technology that can connect a massive number of smart devices around the world to each other and to the Internet. This ability can be used for applications in smart homes, smart cities, healthcare, and industry. The key enablers of IoT were advances in communication in Wireless Sensor Networks (WSNs) and the increasing availability of low-cost and low-power devices that led to the popularity of IoT networks. However, these devices are constrained with limited resources, including memory, power, and storage, making them an attractive attack target. Providing security is a challenging task in IoT. In software maintenance of IoT devices, vulnerabilities first need to be identified and assessed. Then, a lightweight and secure mechanism need to be used to deploy upgrades and patches regularly. Other than security reasons, a firmware update can also be utilized to deploy new features to the devices. In IoT, updates are usually performed Over-The-Air (OTA), eliminating the need for detaching devices and attaching cables during the update procedure, further reducing maintenance costs. There are many challenges in IoT environments that make secure OTA upgrade procedures difficult. One of the main challenges is that IoT devices are resource-constrained; furthermore, communication links often have low reliability and low bandwidth. This challenge makes it difficult to apply security solutions, such as implementing support for secure network protocols or digital signatures. However, many secure and lightweight communication protocols were designed for IoT at different network layers; some application layer protocols are CoAP, MQTT, and OSCORE. These protocols consider the limitations of IoT devices and can cope with low-bandwidth channels in IoT networks. Since IoT devices are often battery operated, the devices' lifetimes are highly affected by the running operations and used protocols. An upgrade procedure is a highly energy-consuming task (due to writing data to the flash memory); therefore, according to the network and devices' requirements, the most suited protocol and security options (considering energy consumption) for OTA upgrade need to be selected or designed.

In IoT low-bandwidth networks, instead of flooding the network with broad-

cast messages, multicast transmission is preferred, especially in OTA update cases where the update or patch targets a large number of devices. In multicast OTA upgrade, a group of IoT devices can be upgraded at once without flooding the network. This requires an efficient selection of devices that need to be upgraded. After selecting or grouping the devices for upgrade, one of the challenging tasks is to share a secret key among group members (devices) that can be used later to protect upgrade messages. The secret key can be transferred to the devices (belonging to the group) in a secure one-to-one way, but even in the key establishment phase in IoT environments, a one-to-many way is preferred.

Another challenge in IoT is heterogeneity. The devices are heterogeneous and are based on different network platforms (including both resource-constrained devices and resource-rich servers) and protocols. Meanwhile, they need to interact with other devices or back-end servers through different protocols. Interoperability is used in IoT to address heterogeneity challenges; interoperability is the ability to enable heterogeneous devices and servers to communicate with each other in an efficient way and exchange data. Interoperability is a critical aspect of IoT, and without interoperability, it is challenging to enable IoT networks' connectivity. In IoT, interoperability is a complex and difficult aspect since solutions to enable interoperability need to consider heterogeneity of IoT environments, including protocols, communication technologies, standards, etc.

Traditional solutions to provide interoperability through intermediary elements such as physical gateways are not scalable, and usually have poor security [Hee+11]. Therefore, new scalable and secure solutions to enable interoperability for IoT are required. To facilitate interoperability in IoT, Software Defined Network (SDN), which is a set of networking technologies, can be used. SDN networking abstracts and virtualizes network functionalities. The administrator can access and control network functions through the SDN controller. SDN can be used to make IoT interoperability scalable and abstracted from the endpoints. To deal with interoperability security, middle-boxes (that perform protocol translation) can be created within Trusted Execution Environments (TEEs). TEE is a secure processing environment where the integrity and confidentiality of the loaded code and data can be protected. An unauthorized entity outside the TEE cannot tamper with the code or data inside the TEE. As a result, to enable interoperability through protocol translation, the protected transferred data can be decrypted, processed, and re-encrypted securely inside a TEE.

Cloud computing provides on-demand system resources, data storage, and computational power, and it has several benefits, including scalability and (almost) unlimited resources. Cloud computing can be integrated with IoT networks to fill the gaps of IoT, such as limited storage and limited computational power. SDN is used in cloud computing to enable efficient network configuration and improve network performance. Public key cryptography, rather than symmetric cryptography, is mainly used to enable secure networking in SDN and cloud computing. Public key cryptography is scalable; however, it requires heavy operations that

make it unsuitable for most IoT environments [TTW17]. As a consequence, symmetric key cryptography is preferred in IoT. Symmetric key operations require less computational power, but key provisioning is more difficult than in public key cryptography. SDN can be used as a tool to efficiently provision and manage symmetric keys in cloud and IoT environments.

In this thesis, we first focus on identifying and assessing vulnerabilities during software maintenance. Then we focus on analyzing and designing security practices and solutions that can be utilized by IoT devices to make secure and lightweight upgrades feasible. These solutions consider the challenges of IoT environments mentioned above. Later we focus on designing a solution that makes secure interoperability possible between IoT protocols. Finally, we focus on developing key management and distribution mechanisms suitable for IoT networks. Our solutions utilize tools and approaches including SDN, TEE, and cloud computing. They are designed for different stages of the software upgrade procedure, including the key establishment and the actual update.

1.1 Dissertation Outline

After the brief introduction and the scope of this dissertation, the rest of the dissertation is organized as follows. In Chapter 2, we present the technical background together with the current state of research topics in this dissertation.

In section 2.1, we describe software maintenance and a tool used for software maintenance (maturity model). In section 2.2, we discuss IoT concepts and common application layer protocols in IoT. In section 2.3, we cover cryptographic concepts such as symmetric cryptography, asymmetric cryptography, hash functions, digital signatures, etc. In section 2.4, we explain transport layer security protocols, including TLS, DTLS, and TLS extensions. In section 2.5, we present the concept of Over-The-Air (OTA) updates in IoT, transmission ways for OTA updates, security requirements of OTA updates, and attacks on OTA updates. In section 2.6, we describe group key management and different approaches for that. In section 2.7, we explain the concept of interoperability in IoT, interoperability approaches, and TLS interception. In section 2.8, we discuss cloud computing and the technologies enabling cloud computing, such as virtualization and orchestration systems. In section 2.9, we cover SDN networking, OpenFlow protocol, and Open vSwitch. In section 2.10, we describe the concept of Trusted Execution Environment, Intel SGX, attestation, and some other Trusted Execution Environments. In section 2.11, we explain formal verification, modeling, and verifying security properties using ProVerif. We present the contributions of each paper in section 3.1, that is followed by conclusions in section 3.2. Finally, in the second part of this dissertation, we present the included publications.

Background

In this chapter, we present the background of the various research fields of the dissertation contributions.

In the following section, we explain software maintenance and maturity model; it plays an important role in software vulnerability handling.

2.1 Software Maintenance

The Software Development Life Cycle (SDLC) [GS15] includes different stages such as design, development, testing, deployment, and maintenance. Maintenance is in the last stage of SDLC, and IEEE [Com+90] defines software maintenance as "*the process of modifying a software system or component after delivery to correct faults, improve performances or other attributes, or adapt to a changed environment*". Vulnerability handling is an essential part of software maintenance. Security vulnerabilities [SZ12], weaknesses in software design or implementation that can be exploited and result in software security breaches, can be seen as a type of fault in software that has gained more attention in recent years. Security vulnerabilities can cause attacks impacting organizations and customers on a large scale. In recent years due to the increasing demand for IoT networks and cloud computing, the attacks have become more sophisticated. Hence, an important part of software maintenance is regularly applying software updates to eliminate known security vulnerabilities. The main focus of this thesis is software updates in IoT networks, and in such networks, the devices are usually unattended for long periods. They are required to be frequently re-programmed Over-The-Air (OTA) to resolve security vulnerabilities. OTA updates and its challenges (addressed in this thesis) are described in detail in section 2.5.

2.1.1 Software Components

Software development can be component based such that reusable components are assembled and developed. These components are usually divided into two types [BWP16]:

- **Open Source Software (OSS):** OSS components are obtained from different OSS communities, and instead of developing all the code in-house, they can be used to decrease development and software maintenance time. OSS software component maintenance can focus on updating the component when new versions are released by the community.

- **Commercial Off-The-Shelf (COTS):** COTS components are available in commercial sources, and they are usually bought from those sources or companies. In COTS components, time to market and also the development time and costs are much lower than in-house developed components [Wel08].

Identifying and evaluating new vulnerabilities accurately and efficiently and regularly applying updates are important. It is also urgent to apply updates resulting from security vulnerabilities or breaches in organizations using OSS or COTS components.

Maturity models can help organizations in the software maintenance phase to identify and handle vulnerabilities; maturity models are explained in detail in the following section.

2.1.2 Maturity Model

Maturity is defined as "*a measure to evaluate the capabilities of an organization in regards to a certain discipline*" [De +05]. Maturity models outline the desired, and logical evolution paths towards maturity [BKP09]. They are seen as tools that facilitate internal and external benchmarking and help organizations improve their way of working by introducing and implementing organizational changes. These changes are a step-by-step progression towards organizations' capabilities [Bec+10]. Implementing and applying maturity models is a slow process and requires resources, efforts, and support both from management and other workforces throughout the organization. In order to support the required changes for implementing improvements, the internal communication processes must be well defined and implemented. Organizations can use maturity models to identify the issues in improving and prioritizing their efforts (for example, in vulnerability handling). It also helps organizations to make sure no important aspects in implementing the changes are neglected. However, maturity models do not describe how these changes should be implemented; hence these changes are highly dependent on the type, size of the organization, business domain, regulations, etc.

In order to assess the maturity of organizations, various maturity levels can be used, and organizations need to fulfill some particular characteristics to reach a specific level of maturity [Bec+10]. The maturity levels provide information about the state of an organization. Different maturity models are described with 4 or 5 maturity levels, 5 maturity levels were defined in [Tea02] which are the basis of many other maturity models, and they are described below:

- **Level 0**: At this level, the processes within the organization are either not performed or partially performed, but the defined goals of the processes are not satisfied.
- **Level 1**: The specific goals of the enterprise are satisfied at this level. At this level, the required work and practices to produce identified output from specific input are supported. However, these practices might be performed informally without following documented plans.
- **Level 2**: At this level, the processes are managed, which means that they are performed based on defined policies using available resources to produce output. The processes are monitored, controlled, reviewed, and evaluated. Based on the results of the output, corrective actions are performed.
- **Level 3**: Processes are defined, established, and improved over time at this level. The organization establishes standard processes that are the base elements of defined processes. Standard processes also describe the relationship between these elements.
- **Level 4**: At the highest level, statistical and other quantitative measures are used to control managed processes. Quantitative objectives are also defined and used as criteria to manage the processes. These objectives are based on the capability of the organization to set standard processes, the organization's business objectives, and customers' need, etc.

Today, a variety of maturity models are used by organizations, and they contain some aspects of vulnerability handling, such as vulnerability identification, assessment of vulnerabilities, and vulnerability disclosure policy. However, vulnerability handling of third-party code (OSS or COTS components) is missing in most of these maturity models. When the use of third-party components increases, it is also important to include handling vulnerabilities of third-party code in the maturity models.

2.2 Internet of Things

The term Internet of Things (IoT) [REC15] is used to describe a network of connected devices or sensors to the Internet. IoT has become popular in recent years in scenarios where connecting to the Internet extends to different sensors, objects, and devices. Objects or devices connected to IoT are seen as key enablers of smart homes, smart cities, and wearables. Connected devices can be monitored and controlled remotely. They can gather and transfer real-time data to other devices or back-end systems for further analysis. These features can improve healthcare monitoring, energy management, traffic control, waste management, and many other things.

Many IoT protocols have emerged to enable Machine-to-Machine (M2M) communication in IoT. They cope with the characteristics of IoT networks, including low bandwidth connections, unreliability, latency, etc. The protocols were also adapted to the limitations of constrained connected devices such as limited power, memory, and CPU [Elh+18]. These protocols have different characteristics, and for each IoT application, the most appropriate protocol should be selected. A sample IoT protocol stack [SG18] is depicted in Figure 2.1. HTTP, MQTT, CoAP, and OSCORE are popular IoT protocols at the application layer. TCP and UDP, as well as TLS and DTLS, are transport layer protocols. IPv6/IPv4 and 6LoWPAN in the network layer provide network routing. At the lowest layer or physical layer, WiFi, BLE or other protocols enable data exchange.

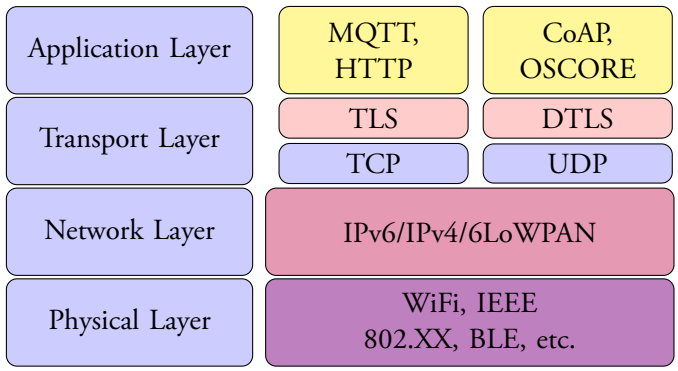


Figure 2.1: A sample protocol stack for IoT

2.2.1 Common Application Layer IoT Protocols

The application layer enables the application communication between IoT endpoints. It is responsible for providing services and determining a set of protocols for message exchange at the application level [YS+16]. In this section, we explain the common IoT application layer protocols related to this thesis.

HTTP

Hyper Text Transfer Protocol (HTTP) was developed in 1997 by IETF and W3C jointly [Nai17]. It is a request-response protocol in client-server networks; it is a text-based protocol that uses the Universal Resource Identifier (URI) instead of using topics to send and receive data [Nai17] (topic is a string that is used in MQTT to tag and filter messages). HTTP is a stateless protocol since each request is handled independently without knowledge of previous requests. HTTP typically uses TCP as the underlying transport protocol and TLS for security, but HTTP does not provide Quality of Service (QoS).

CoAP

Constrained Application Protocol (CoAP) is a lightweight protocol and was developed by IETF CoRE (Constrained RESTful Environments) Working Group. It utilizes a URI to identify resources available on IoT devices instead of topics. CoAP is similar to HTTP, with some modifications to meet IoT-specific requirements, such as operating in Lossy Networks (LN) and Low Power (LP) consumption [BCS12]. It is a request and response protocol that runs over UDP; thus it supports both unicast and multicast CoAP requests. In CoAP reliability is provided by the type of the messages and messages can be *confirmable* or *non-confirmable*. Confirmable messages always require an acknowledgment (ACK) by the receiver. In contrast, non-confirmable messages do not, and this type of message is useful when the messages repeat regularly for the application requirements. Like HTTP, CoAP uses different methods, including GET, PUT, POST, and DELETE. CoAP uses datagram transport layer security (DTLS) to protect the network communication [SHB14b], and CoAP supports four different security modes:

- **NoSec:** DTLS is disabled in this mode, and no protocol-level security is provided.
- **PreSharedKey:** In this mode, DTLS is enabled, and there is a list of pre-shared keys, and each of them includes a list of nodes that the key can be used to communicate with.
- **RawPublicKey:** In this mode, DTLS is enabled, and an asymmetric key pair without a certificate (a raw public key) is pre-installed on the IoT device.
- **Certificate:** DTLS is enabled and the device uses asymmetric key pair but this time an X.509 certificate is used that is signed by a trusted root. The device has a list of trusted roots in order to validate a certificate.

In order to avoid IP fragmentation, a CoAP message should fit into a single IP datagram; hence an IP Maximum Transmission Unit (MTU) of size 1280 bytes should be used. If nothing is known about the headers, an upper bound of 1152 should be set for the message size, and 1024 bytes for the payload size [SHB14b].

For scalability and efficiency in CoAP, proxies can be used, but CoAP-to-CoAP or CoAP-to-HTTP proxies require the termination of DTLS or TLS session at the proxies [Sel+19b]. Therefore, there is a security risk in the sense that an untrusted proxy might eavesdrop and manipulate the payload of the messages. One way of avoiding the end-to-end security problem with proxies is to use object security that protects the messages on the application level, i.e., the CoAP level. This was the primary motivation for introducing the Object Security for Constrained RESTful Environments (OSCORE) protocol [Sch17].

OSCORE

OSCORE is an extension of the CoAP protocol, and instead of securing a session, it aims to protect end-to-end CoAP requests and responses. OSCORE protects the messages on an object level and uses the Concise Binary Object Representation (CBOR) for compact encoding. The messages are encapsulated as encrypted and authenticated CBOR Object Signing and Encryption (COSE) [Sch17] objects. Besides encryption and integrity protection, OSCORE provides replay protection and request-to-response binding. Request-to-response binding is important to protect against attacks forwarding a response to a request other than the intended one.

OSCORE protects not only the CoAP payload but also the CoAP options and parts of the URI indicating targeted resources in the request. OSCORE is lightweight since it only protects the relevant application layer information, and the added data to the actual CoAP message is 11-13 bytes. In order to use OSCORE, the devices must support CBOR and COSE, as well as the HMAC-based Key Derivation Function (HKDF) and Authenticated Encryption with Associated Data (AEAD) algorithms for key derivation and authenticated encryption. Other than that, the devices must have an OSCORE security context to extract the keying material [Gun+21]. The security context is a set of the necessary information to perform cryptographic operations. Sender and recipient contexts are derived from the common context. The sender protects messages by sender context, and the receiver verifies them using recipient context. To derive the security context, OSCORE requires a pre-shared master secret on the devices.

In OSCORE, different parts of CoAP messages can be protected in different ways [Gun+21]. Confidential data is encrypted and integrity protected and cannot be read or tampered with by the intermediate nodes such as proxies. Static data is only integrity protected and can be read, and dynamic data is not protected at all.

MQTT

Message Queue Telemetry Transport (MQTT) is a messaging publish/subscribe protocol over TCP. It is a lightweight protocol and can be used for many-to-many communications [SM17]. In MQTT, a broker (server) operates alongside a set of clients. The broker receives messages from the clients and routes them to their designated destination clients. Clients can be publishers or subscribers; publishers publish messages tagged with a topic, and subscribers can receive those messages. The connection between publishers and subscribers is done through the broker, and there is no direct connection between them. MQTT uses TCP as the transport protocol, and to implement communication security, the broker and clients can use TLS for communication. MQTT enables reliability using three different QoS levels [SM17] as described below:

- **QoS0 (At most once):** This level does not provide any acknowledgment of delivering messages, and the messages are sent at most once.
- **QoS1 (At least once):** In this level, it is possible to send messages more than once until an acknowledgment is received.
- **QoS2 (Exactly once):** In this level of QoS, the messages are sent exactly one time with the help of a 4-way handshake to ensure exactly one copy of the message is received.

2.3 Cryptography

The base of secure communication on the Internet is *cryptography* [Kes03], and it is essential for many applications, such as secure payments and protecting passwords. In application-to-application communication over the Internet, cryptography needs to be used in order to provide the following features [Kes03]:

- **Confidentiality:** To make sure no one except the intended receiver can access the message.
- **Integrity:** To make sure that the message has not tampered.
- **Authentication:** To prove the identity of communicating peers.

In general cryptographic schemes are divided into two groups [Sti05], symmetric key cryptography and asymmetric or public key cryptography; and they are further discussed below.

2.3.1 Symmetric Key Cryptography

In symmetric key cryptography, a symmetric secret key is used both for encryption and decryption, and the same symmetric key must be known by both sender and receiver [Kes03]. Thus, the major concern in symmetric key cryptography is key distribution, or key management [Blu+92]. Symmetric methods are generally categorized into two categories: *stream ciphers* or *block ciphers*. In stream ciphers, each byte is encrypted at a time, while in block ciphers, one block of data is encrypted at a time. In this thesis, we only used block ciphers, and we describe the used block cipher modes next:

- **Cipher Block Chaining (CBC):** In this mode, each plaintext block is XORed with the previous ciphertext block before encryption [Kes03]. Thus, the ciphertext blocks are dependent on each other. The first block is XORed with an initialization vector (IV).

- **Galois/Counter (GCM):** GCM [MV04] combines Counter Mode (CTR) of encryption with the Galois message authentication code (GMAC), GMAC can generate incremental message authentication codes (message authentication codes are described in section 2.3.4). Similar to CTR, a counter for each block is considered, and then this block counter is combined with an IV. The combined value is then encrypted, and the result is XORed with the plaintext to produce each block of ciphertext.

- **Counter with Cipher block chaining Message authentication code (CCM):** This mode combines CBC-MAC mode (CBC-MAC of a message is generated by encrypting the message in CBC mode with zero IV and keeps the last block.) with the CTR mode of encryption [WHF03]. These modes are applied in authenticate-then-encrypt order. This means CBC-MAC is applied to the message first to generate a tag; then, the message and the tag are encrypted together using CTR mode. The difference between CCM_8 with CCM is that it uses 8 octets for authentication instead of 16 octets.

One of the most popular symmetric key cryptography algorithms is Advanced Encryption Standard (AES) [Dwo+01], which is used in this thesis. AES was established by the National Institute of Standards and Technology (NIST) in 2001. In AES, three key lengths can be used [Kes03]: AES-128, AES-192, and AES-256; key lengths of 128, 192, and 256 bits are used to encrypt and decrypt 128 bits block of messages, respectively. The key size specifies the number of transformation rounds that should apply to the input to generate the final output. The required rounds on different key lengths are as follows: 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys. In AES, before applying the rounds, a plaintext input is converted into four to four matrices consisting of rows and columns. Each round is then built up of several processing steps [SV18]: substitution, transposition, and mixing of the columns of the plaintext input. In the substitution step, each byte of data is replaced with another byte according to a substitution table. In the transposition round, the bytes in each row are shifted to the left a certain number of times. In the mix columns step, each column consisting of four bytes is multiplied with a defined matrix. Finally, the resulted matrix is XORed with the key to generate the cipher text.

2.3.2 Asymmetric Key Cryptography

In asymmetric encryption, a pair of keys, public and private keys, are used to encrypt and decrypt data [Kes03]. The public key may be known by others and is used for encrypting messages. Then, the relevant private key is used to decrypt the data. Diffie-Hellman (DH) [Hel+76] algorithm was one of the first development in the area of public key cryptography, it was discovered in 1976 and was invented as a solution to the key distribution problem. In the DH algorithm, both sender and receiver require to have private and public key pairs. Both peers compute the

same shared secret by combining one's private key with the other peer's public key [Res+99]. For instance x, y, $A = g^x \ mod \ p$, $B = g^y \ mod \ p$, are the private keys and public keys of sender and receiver respectively (p is a large prime and g is a primitive root modulo p). Then, the shared secret S can be computed by both peers as: $S = B^x \ mod \ p = A^y \ mod \ p = g^{xy} \ mod \ p$. The security of DH algorithm is based on hardness of discrete logarithm problem (DLP) that for a given p, g, and $g^x \ mod \ p$ it is hard to compute x.

One of the most well-known public key cryptography algorithms is RSA (Rivest –Shamir–Adleman) [RSA78] that is used mostly for digital signatures (digital signatures are explained in detail in section 2.3.5) or encryption of small blocks. The public private key pair is derived from a large number, *n*, which is the product of two prime numbers *p* and *q* (*n* is used as the modulus for both the public and private keys). The exponent *e* is selected that is a number between 1 and *(p - 1)(q – 1)* and the only common factor between *e* and *(p - 1)(q – 1)* is 1. The public key is then the bundle of *(n,e)*. The number *d* is then derived from the inverse of *e* modulo *(p - 1)(q – 1)*. The private key is bundled as *(n,d)*. To encrypt a message, M, the sender creates the cipher text as $C = M^e \ mod \ n$. The receiver then decrypts the cipher text using the private key as $M = C^d \ mod \ n$. In these calculations, the attacker cannot determine the prime factors of *n* or, subsequently, the private key. The security of RSA is due to the difficulty of factorization of large prime numbers.

Elliptic Curve Cryptography (ECC) is another public key cryptography that was first proposed independently by Koblitz [Kob87] and Miller [Mil85] in 1985. ECC is based on elliptic curves over finite fields and can offer the same level of security as RSA but with much smaller key sizes. This makes it ideal for devices with limited resources, including power and memory, e.g., constrained IoT devices and smart cards [Kes03]. The security of the ECC algorithm is based on the Elliptic Curve Discrete Logarithm problem that is hard to solve [KAS08]. An Elliptic Curve (EC) is defined as $y^2 = x^3 + ax + b$. One of EC features is addition; adding two points on the curve result in another point that is again on the curve. Another EC feature is multiplication, multiplying a positive integer *k* to a point result as the sum of *k* copies of the point. The EC can also be used in DH key exchange protocol, and the resulting protocol is called Elliptic-Curve Diffie–Hellman (ECDH) [BJS07]. In the ECDH key exchange protocol, the sender first chooses a random secret d_a that is its private key and then generates its public key as $Q_a = d_a G$, where G is the base point on the curve that sender and receiver agreed on. The receiver does the same procedure and calculates its public and private key pair. After exchanging the public keys, the sender and receiver can calculate the shared secret *S* as [KAS08]: $S = d_a Q_b = d_a d_b G = d_b d_a G = d_b Q_a$ and use it to secure further communication.

Different ECC curves have different names (e.g., secp256r1, secp256k1), field size (that defines the key length, e.g., 256 bits), strength, and other parameters. ECC keys can have different length such as: 192-bit (curve secp192r1), 224-bit

(curve secp224k1), 256-bit (curves secp256k1 and secp256r1), etc. The main curves used in this thesis are the Koblitz curve (such as secp256k1) and the prime field curve (such as secp256r1). The main difference between them is that Koblitz curves are a few bits weaker than prime field curves [Bjo09]. Prime field curves are NIST curves and are the most widely used.

2.3.3 Hash Functions

Hash functions or message digest algorithms are one-way mathematical functions that take an input and generate a fixed length output without the usage of any key [Sti05]. In hash functions, the input is called the message, and the output is called the hash value or digest. Cryptographic hash functions have additional security requirements and are used in security applications. Cryptographic hash functions are not reversible, meaning it is impossible to obtain the plaintext or its length from a hash value. The usage of cryptographic hash functions is to provide a *digital fingerprint* on the data to ensure that an intruder attacker has not tampered with it. The widely used hash algorithms belong to the Secure Hash Algorithm (SHA) family.

SHA is a family of cryptographic hash functions published in 1993 by the U.S. government agency NIST [SA93]. SHA hash functions include SHA-0, followed by SHA-1 (1995), SHA-2 (2001), and SHA-3 (2015). In this thesis, SHA-2 is used. SHA-2 consists of different hash functions, such as SHA-224, SHA-256, SHA-384, and SHA-512. SHA-256 with 32 bits output and SHA-512 with 64 bits output are widely used hash functions. These two hash functions have a similar structure but differ in the number of rounds, shift amounts, and additive constants. The SHA-2 family is considered to be secure in most applications.

The family SHA-3, which is the latest member of the SHA family, was released by NIST [Dwo+15] in 2015. It consists of four cryptographic hash functions: SHA3-224, SHA3-256, SHA3-384, and SHA3-512. SHA-3 is based on Keccak, which was designed by Bertoni et al. Keccak was developed in 2012 as a fully functional hash standard. Later in 2014, NIST published the SHA-3 draft as FIPS 202, and finally, it was approved in 2015 [Dwo+15]. The SHA-3 uses a sponge construction in which the data is absorbed into the hash function, and then the result is squeezed out [DCD17]. In the absorbing phase, small blocks of the data are mixed into the buffer using XORs; then, the result is transformed as a whole using a permutation function. Later in the squeezing phase, instead of XORing, the same blocks of data are extracted as the output.

2.3.4 MAC

Message Authentication Codes (MACs) are known as tags and are used between two parties that share a secret key to protect the integrity of transmitted messages [Tur08]. The authenticity of the messages is also protected using MAC to ensure the message was sent from the actual sender. Similar to hash functions, a MAC

function also compresses an input into an output with a fixed length. The main difference between the hash and MAC functions is that MAC utilizes a secret key in the compression procedure. MACs are different from digital signatures since, in the MAC, the output values are generated and verified with the help of the same secret key. Therefore, the sender and receiver must agree on a key before transferring messages (similar to symmetric encryption).

HMAC or Hash-based Message Authentication Code [Tur08] is a specific type of MAC which utilizes a hash function and a secret cryptographic key to obtain the HMAC value.

2.3.5 Digital Signatures

Digital signatures are used in many applications today to verify that the sender of the message is authenticated (authenticity) and to ensure that the message has not been tampered with after signing (integrity). They can also provide a non-repudiation feature, which means that the signer of the message cannot later claim they did not sign the message. It is used to prove that the alleged sender actually sent the message. Digital signatures utilize asymmetric cryptography. Generally, a digital signature scheme consists of three main steps: 1) key generation where the private and public keys are selected; 2) signing step in which a signature is created using a message and the private key; 3) verification step where the signature is verified using the message, the public key, and the signature. The digital signature algorithms used in this thesis are described below:

- **RSA**: The RSA public key cryptography can also be used for digital signatures to sign and verify messages. RSA digital signatures are based on the modular exponentiation and the computational difficulty of the RSA problem [Sti05]. Similar to RSA cryptography, the RSA signature algorithm key pair consists of a public key *(n,e)* and a private key *(n,d)*. RSA signing and verifying are as follows: 1) The hash of the message is generated as: h = *hash(message)*; 2) The hash is used to create the signature as: $signature = h^d \ mod \ n$, the created signature is then in range *[1..n]*. 3) To verify the signature, the hash of the message (h) is generated, and h' is calculated as: $h' = signature^e \ mod \ n$, if h and h' are equal then the signature is valid.

- **DSA**: The Digital Signature Algorithm or DSA was proposed by NIST in 1991 [Nis92]. DSA is also based on modular exponentiation, discrete logarithm problem, and its computational difficulty. Similar to RSA, this algorithm uses a public and private key pair, the private key is used for signing a message, and the signed message is verified by using the signer's public key. The DSA algorithm consists of three steps [Nis92]: 1) in key generation, first, a prime number q, known as the prime divisor, is selected. Then another prime number, p, is selected such that *p-1* is a multiple of q.

Then the integer *h* is selected randomly from *[2..p-2]*, and *g* is calculated as: $g = h^{(p-1)/q}$ *mod p*. *x* is the private key which is a random integer from *[1..q-1]* and *y* the public key that is $y = g^x$ *mod p*. Finally the private key is packaged as *{p, q, g, x}* and the public key package is *{p, q, g, y}*. 2) The message is first hashed to get the hash value H in the signature generation. A random integer *k* is selected from *[1..q-1]*, then *r* is calculated as: $r = (g^k$ *mod p) mod q*. Then *s* is calculated as: *s* = k^{-1}*(H+xr) mod q*. The signature is then *(r, s)*. 3) In signature verification, first the hash of the message is calculated (H) and it should be verified that $0 < r < q$ and $0 < s < q$. Then the following values are computed: w = s^{-1} *mod q*, $u_1 = Hw$ *mod q*, $u_2 = rw$ *mod q*, $v = (g^{u_1}y^{u_2}$*mod p) mod q*. If *v* and *r* are equal the signature is valid.

- **ECDSA**: The Elliptic Curve Digital Signature Algorithm (ECDSA) [JMV01] is a variant of the DSA that uses elliptic curve cryptography. For creating the ECDSA signature sender and receiver first must agree on curve parameters *(Curve, G, n)*. The hash of the message is calculated as *e*. The private and public key pair is *($d_a, Q_a = d_aG$)* respectively. A random integer *k* is selected from *[1..n-1]*, and a curve point is calculated as *(x_1, y_1)* = *kG*. Then *r* and *s* are calculated as *r* = x_1 *mod n*, *s* = $k^{-1}(z + rd_a)$ *mod n* where *z* is the L_n leftmost bits of *e*. The signature is then *(r, s)*. The receiver then first verifies if *r* and *s* are in the range of *[1..n-1]*, then the hash of the message is calculated (*e*) and the L_n leftmost bits are extracted as *z*. Then, u_1 = zs^{-1} *mod n* and u_2 = rs^{-1} *mod n* are calculated. Finally the curve point is calculated as *(x_1, y_1)* = u_1G + u_2Q_a, if *r* = x_1 *mod n* then the signature is valid.

2.3.6 Elliptic Curve Integrated Encryption Scheme

In this thesis, other than public key and symmetric key cryptography, we used Integrated Encryption Scheme (IES). IES provides capabilities for encryption, key exchange, and digital signatures; hence, it is called Integrated Encryption Scheme. IES is a hybrid encryption scheme that uses both public key and symmetric key cryptography, and the security of this scheme is based on the computational Diffie–Hellman problem [JN03]. A variant of IES used in this thesis is Elliptic Curve Integrated Encryption Scheme (ECIES). ECIES uses the following functions [MEÁ10]:

- Key Derivation Function (KDF) produces a set of keys from keying material and other optional parameters.
- Symmetric encryption algorithm.
- MAC is used to authenticate messages.

The elliptic curve parameters are *(p, a, b, G, n, h)*, where *p* is the prime number, *a* and *b* are parameters defining the curve, *G* is the base point on the curve, *n* is the order of elliptic curve, and *h* is the cofactor.

The encryption steps are as follows [MEÁ10]: 1) Alice or the sender generates a random number *r* in the range *[1..n-1]* and calculates *R = rG*. 2) Then, a shared secret is derived as $S = P_x$, where $P = (P_x, P_y) = rK_B$ (K_B is the public key of Bob derived from Bob's private key or k_B as $K_B = k_B G$). 3) KDF is used to derive the symmetric encryption and MAC keys as $k_E||k_M$ = *KDF(S)*. 4) The message is encrypted as *c = E(k_E,m)*. 5) The tag is calculated as *d = MAC(k_M,c)*, and finally the output message is *R||c||d*.

Upon receiving the message Bob follows these steps to decrypt the message: 1) Bob derives the shared secret $S = P_x$ using $P = (P_x, P_y) = k_B R = k_B rG = rK_B$. 2) Then he derives keys using *KDF* the same way Alice derived the keys. 3) He checks whether *MAC(k_M,c)* is equal to *d* or not. 4) If the MACs are equal, he uses symmetric encryption to decrypt the message.

2.3.7 Quantum Resistant Cryptography

Advances in quantum computing can break classical cryptographic systems, which are based on the hardness of computational problems such as integer factorization and discrete logarithms. These problems can be solved efficiently by a quantum computer [Sho94]. If a quantum computer with several thousands quantum bits can be built, many public key encryption algorithms, including RSA and ECC, will no longer be secure [Sho94]. However, the impact of quantum computers on symmetric key cryptography is not as drastic as public key encryption. To protect against Grover's algorithm [Gro96a], which provides a quadratic speed-up for quantum search algorithms, usually doubling the key size will be sufficient to preserve security [Che+16]. Exponential speed up for search algorithms, including Grover's algorithm, is not possible, which makes symmetric algorithms and hash functions usable in the quantum era as well [Ben+97]. Another countermeasure is to switch to computational problems that are hard to solve with quantum computers. These problems are *quantum-resistant* including discrete lattices [MR09] and hard coding problems [OS09]. These algorithms are out of the scope of this thesis.

2.4 Security of Transport Layer

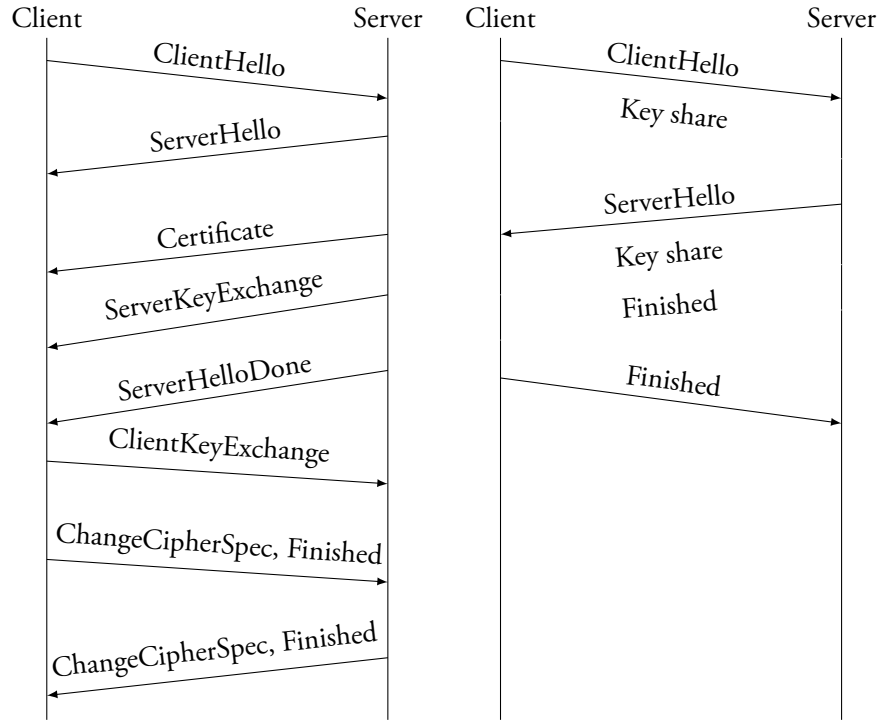
Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) are designed to provide communication security at the transport layer [DR08a]. The main goal of TLS/DTLS is to provide data privacy and integrity between communicating peers, for that two communicating peers establish a secure *session* and use it to protect the application messages. The main difference between DTLS and TLS is that DTLS is built on UDP, while TLS uses TCP. TLS and DTLS

protocols are composed of two layers: record protocol and handshake protocol [DR08a]. The handshake protocol is used to authenticate the communicating peers, negotiate cryptographic information, and establish a shared secret. While the record protocol is used to protect traffic between the communicating peers with the help of established parameters during the handshake protocol. In the following, we explain these protocols and their differences in different versions of TLS and DTLS protocols.

2.4.1 TLS

In the record protocol of TLS, the connection has the following features [DR08a]; it is encrypted with a symmetric encryption (e.g., AES), and for each separate connection, these keys need to be generated uniquely. The connection is reliable, and the messages include an integrity check using a keyed MAC (e.g., SHA-2 and SHA-3). The TLS record protocol is used to encapsulate higher-level protocols such as the TLS Handshake Protocol. In the TLS handshake, first, the client and the server specify which version of TLS they use and decide on cipher suites, a set of encryption algorithms to establish the secure connection they are going to use. Then, the client and server authenticate each other using asymmetric public key encryption (e.g., RSA), and they generate session keys to use as symmetric encryption after finishing the handshake. Another possible approach for authentication during the handshake is using symmetric pre-shared keys [ET05a] owned by the client and the server. In this approach, the client suggests to the server which symmetric key it intends to use. TLS 1.2 handshake protocol is depicted in Figure 2.2a, and it is built up of the following steps [DR08a]:

- First, the client sends ClientHello message to the server, then the server sends back a ServerHello message. In the hello messages, the TLS protocol version, session ID, cipher suites, and generated random values are exchanged.
- Following the hello messages, the server sends its certificate back to the client to be authenticated. The client verifies the server certificate and confirms the server is the actual server. The server may send a ServerKeyExchange message only if required (e.g., if the server has no certificate). The server may require the client certificate as well. When the hello messages are complete, the server sends a ServerHelloDone message to the client.
- Based on the public key algorithm selected between the ClientHello and the ServerHello messages, the client sends the ClientKeyExchange message, which contains secret key information encrypted using the server's public key. The server decrypts the message with its own private key. Both client and server can now generate the secret key (session key) based on the received information.



(a) TLS 1.2 handshake **(b)** TLS 1.3 handshake

Figure 2.2: TLS 1.2 and 1.3 handshakes

- Finally, the client sends the ChangeCipherSpec message followed by a Finished message encrypted with the secret key. The server also sends back its ChangeCipherSpec message followed by a Finished message encrypted with the secret key. At this step, the handshake is done, and the client and server can exchange application layer data.

TLS 1.3 [Res18] is the latest version of TLS, which was defined by Internet Engineering Task Force (IETF) in 2018. The main differences between TLS 1.2 and TLS 1.3 are as follows: 1) in TLS 1.3, a zero round-trip time (0-RTT) mode was added to save a round trip during the connection. Using 0-RTT, the client can send data early on the first flight (in case multiple messages are transmitted at the same time, they are grouped together as a *flight*). 2) TLS 1.3 mandates forward secrecy, and a new session key is generated for each new session; even if a session gets compromised, the adversary cannot gain access to the previous messages. TLS 1.3 handshake has three phases [Res18] that are depicted in Figure 2.2b and these phases are:

- The first phase is the key exchange phase, and in this phase, the client sends

the ClientHello message containing a random value, supported protocol versions, a list of supported cipher suites, and the client's key share for the key agreement protocol. The client may include the PSK extension if a pre-shared secret has been established between the client and the server. This extension is explained in section 2.4.3.

- The server then sends back its own ServerHello message, which contains the selected key agreement protocol and the server's key share, as well as the server Finished message (In TLS 1.3, sending the Finished message already in step 2 saves 4 steps compared to TLS 1.2).

- The client can generate the secret key based on the server's key share, and finally, the client sends back a Finished message. At this point, the handshake is complete, and the client and server can send encrypted application data.

2.4.2 DTLS

TLS is the most widely used protocol for securing network traffic; however, it must run over TCP and cannot be used to run over unreliable transport protocols, e.g., UDP. DTLS [RM+06] was designed to execute over UDP, and it is a datagram compatible version of TLS. DTLS does not provide any reliability, and it is suitable for use in applications such as online gaming, media streaming, etc.

The record layer in DTLS is quite similar to TLS; only an explicit sequence number is added to the record of each DTLS message. This sequence number makes the messages independent from one another; thus, they can be correctly processed despite the unreliable transport protocol and out-of-order delivery. The sequence number allows the recipient to verify the DTLS MAC. DTLS MAC handling differs from TLS, and records with bad MACs are just discarded, while in TLS, the connection is terminated. The other difference between DTLS and TLS is that stream ciphers are not supported in DTLS. DTLS record must fit into a single datagram to avoid IP fragmentation. Thus, DTLS messages contain a fragment offset and a fragment length.

The DTLS handshake is depicted in Figure 2.3. The client starts the handshake with *ClientHello* message. Once the ClientHello is received by the server, a *Hello Verify Request* which includes a locally generated *cookie* is sent to the client [TGS17]. After receiving this message by the client, it should respond with the second ClientHello message that includes the cookie from the server. The server verifies the cookie, and only if it is valid it proceeds with the rest of the handshake.

DTLS handshake messages are similar to TLS handshake with three main changes:

- A stateless *cookie* is added to prevent Denial-of-Service (DoS) attacks. After the ClientHello message, the server may respond with a Hello Verify

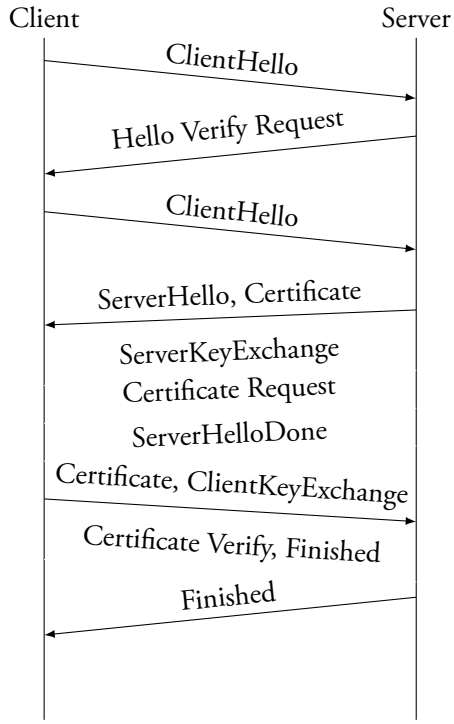


Figure 2.3: DTLS handshake

Request message that contains a stateless cookie and causes the client to resend the ClientHello with the added cookie. This technique makes DoS attacks with spoofed IP more difficult, since it forces the attacker to receive a cookie.

- In order to avoid IP fragmentation, the handshake message is divided into N different parts with the same sequence number, and the fragment offset in these messages is set.
- A retransmission timer is added to handle message loss. In DTLS, messages are grouped together as message flights, and in case the timer expires, the whole flight of messages will be retransmitted. Packet loss can be handled by means of local timeouts and message retransmission policies [TGS17].

2.4.3 TLS Extensions

TLS messages can contain different extensions, and in (D)TLS 1.3, in case PSK is used to establish cryptographic material, the ClientHello message must include *psk_key_exchange_modes* and *pre_shared_key* mode [Res18]. The extension *psk_key_exchange_modes* specifies the *psk_ke* or *psk_dhe_ke* key exchange mode. In *psk_dhe_*

ke or PSK-with-DHE the handshake includes an ephemeral Diffie-Hellman key exchange, which can provide forward secrecy. In ephemeral Diffie-Hellman, a new temporary DH private key is generated for every connection to enable forward secrecy. On the other hand, the *psk_ke* handshake does not provide forward secrecy in the case of PSK.

The *pre_shared_key* extension is used to negotiate the identity of the pre-shared key to be used in a handshake. This extension includes a collection of offered pre-shared keys in the form of a list of key *identities* and key *binders* [Res18], and they are listed as follows:

- **Key identities** are a list of the identities that the client is willing to negotiate with the server.
- **Key binders** are a list of HMAC values, one for each value of the identities in the same order. The HMAC values are HMAC calculated with a binder key derived from the corresponding pre-shared key.

When the client wants to use one of the two PSK-based key exchange modes, first, it suggests one or several PSK identities be used in the *pre_shared_key* extension of the ClientHello message [Res18]. Further, it uses the *psk_key_exchange_modes* extension to indicate which PSK mode is supported such as *psk_ke* or *psk_dhe_ke*. Later, if one of the PSK identities is accepted by the server, the chosen identity will be included in the *pre_shared_key* extension of the ServerHello message.

2.5 Over-The-Air Updates

An IoT system consists of services distributed in various devices that can communicate and collaborate with each other or back-end systems [ROL18]. In such systems, the firmware (software for hardware) is an important part of the system since it interacts with the hardware and it implements the required specifications of the system [FRP17]. Furthermore, it can provide specific functionality in interaction with hardware components. The IoT firmware may change due to system reconfiguration related to the communication with other systems or the users [FRP17]. This change might be required for bug and error correction, mainly due to security-related issues, upgrading the system feature, etc. Applying the changes can be accomplished by changing the firmware without modifying the hardware of the devices. The devices, including IoT devices, require an interface to modify the firmware, and in case this interface is connected to the network, the updates can be applied over the network, which is referred to as Over the Air (OTA) update [FRP17]. Using OTA, the updates can be performed remotely, which decreases the maintenance costs drastically. Automation methods can be applied to the OTA update procedure to apply fixes in a controlled manner. Using OTA, feedback can be sent to the administrator on the success or failure of the update. OTA updates

can be triggered periodically or on-demand. All update procedures throughout this thesis are OTA on-demand updates.

2.5.1 Transmission Ways for OTA Updates

OTA updates can be transmitted in three different ways [Mal+10]:

- **Unicast**: In unicast OTA update, the software or firmware is sent only to one device at a time.
- **Multicast**: In multicast communication, the update is targeted to one or more receivers. In such a case, one or more senders might also be available. IP multicast is a dynamic way of many-to-many connection between a group of senders and receivers. The format of packets in IP multicast is identical to the unicast packets since they use a special class of destination address (indicating a specific multicast group). *TCP* does not support multicast mode; therefore, multicast packets use *UDP* as the transport protocol. In multicast communication, the devices will receive the packets only if they have previously joined the multicast group address [Mal+10]. Group membership can be controlled by the devices or the network administrator. In this thesis, it is only possible by the network administrator.
- **Broadcast**: In broadcast communication, the update is sent to all devices in the network.

Between the above communication methods, multicast is helpful in most IoT networks to perform OTA updates since, in an OTA update, there are usually a group of devices targeted to the update procedure. Multicasting can also save bandwidth [Mal+10], which is limited in most IoT networks.

2.5.2 Security Requirements of OTA Updates

The OTA updates introduce attack vectors, e.g., Bluetooth, and Wi-Fi, to the malicious attackers [HGC20]. Using these attack vectors, the attackers can endanger the OTA updates and further reprogram or control IoT devices. It is essential to provide the security of the OTA updates during the whole update procedure steps, including downloading and storing update packages. Thus, a secure OTA update procedure needs to fulfill the following security requirements [Asc+12; HGC20]:

- **Protecting the update packages during transmission**: Update patches or firmware updates are usually divided into different packages. Authenticity, confidentiality, and integrity of update packages transmitted from the update server to IoT devices must be guaranteed. This protection needs to be established end-to-end from the update server to the end devices.

- **Protecting the update packages while stored**: When update packages are received on the IoT side, they must be stored securely until the software update is installed. It should not be possible for an attacker to tamper with the intermediate packages that are stored.
- **Authorization verification**: In the update or upgrade procedure (update is used to apply patches, while the upgrade is used to update the version), only the authorized servers must be able to update IoT devices.
- **Compromise-tolerant**: In case a single device is compromised, the security of other devices in the network must be preserved.
- **Protection and mitigation of overloading the devices**: In IoT, since the devices' resources are limited, it is possible that an attacker aims to perform a DoS attack. Using a DoS attack, the attacker can force the device to verify the fake or malicious update packages and further cause consumption of resources.

In the majority of IoT applications ranging from smart homes to smart cities, IoT devices are powered by batteries; therefore, DoS attacks targeting these devices are usually battery-draining attacks. The IoT battery draining attacks are explained in section 2.5.3.

2.5.3 OTA Update Attacks

OTA update attacks can have different targets, including IoT devices, intermediate links, and the update servers [EB22]. The attacks on the update servers are out of the scope of this thesis.

The most common attack on the intermediate links is the Man-In-The-Middle (MITM) attack, in which the attacker tries to spoof the exchanged packages between IoT devices and the update server. In a successful MITM attack, the attacker can force the devices to install a tampered or malicious update.

IoT devices are the most resource-constrained devices in the OTA update procedure, and in the network, they are mainly the target of DoS attacks, such as battery drain attacks. Battery drain attacks on IoT devices can be grouped into the following three categories [Smi+20]:

- **Denial of Sleep**: These attacks can cause devices to consume more energy than usual. One example is the jamming attack that causes the network to become unusable. The attacker transfers interfering signals on the network so that the channel will not be able to become idle to receive actual data. This forces devices to retransmit their data repeatedly.
- **Flooding Attack**: In these attacks, the attacker creates massive traffic and sends it to the devices in the network, which causes the receivers to respond to those messages. As a result, the nodes would drain their energy resources.

- **Vampire Attack**: This type of attack does not allow the batteries of devices to stop working by putting a strain on them. An example of such an attack is the stretch attack, where the packet is sent along a longer route in the network by changing the packet's header. This attack can increase the energy consumption by a factor of four [MD14], and it is most likely to succeed in networks where authentication is not performed.

2.5.4 OTA Update Challenges

One of the most blocking challenges in IoT is the limited capability of devices, particularly in performing heavy cryptographic operations [EB22]. However, one of the advancements of IoT devices is to contain *cryptographic hardware acceleration*. Cryptographic hardware acceleration is the use of hardware to perform cryptographic operations faster than in software. However, not all manufacturers include hardware acceleration in their devices due to higher production costs.

Another OTA update challenge in IoT is to efficiently manage key distribution [EB22] and share keys among devices targeting the OTA update. Key management approaches will be further discussed in section 2.6.

One of the other OTA update challenges in IoT networks is interoperability [EB22]; due to the heterogeneous nature of the IoT devices, it is difficult to have a single solution available for all devices. Devices use different applications and protocols (CoAP, HTTP, MQTT, etc.) that are not interoperable. Different solutions exist that can be used in IoT to provide interoperability, and they are presented in detail in section 2.7.

2.6 Group Key Management

The demand for multicast communication is increasing in IoT networks and can be used as an efficient communication mechanism for IoT devices. A secure multicast solution in IoT requires a secure *group key management* scheme. In group key management, there are two main *functional entities* [CS05]: a Group Controller (GC) that is responsible for authentication and authorization and a Key Server (KS) that is responsible for maintenance and distribution of the key material to devices. In this thesis, these two functions are implemented on a single physical entity which is referred to as *server*. In order to multicast a message securely to group members, a secret symmetric key called *session key* needs to be shared among group members. Each group member also has access to a long-term symmetric *master secret* which is used in *re-keying* procedure. Upon joining or leaving a group member, a new session key needs to be created and distributed to provide forward and backward secrecy; this process is called re-keying. The maintenance procedure and the distribution of the secret keys in re-keying are commonly referred to as group key management.

A secure and efficient group key management scheme should fulfill the following requirements [CS05]:

- **Security**: It should provide forward secrecy, meaning that a member should not be able to decrypt the messages after leaving the group. It should also provide backward secrecy; a newly joined member should not be able to decrypt the previously encrypted messages.
- **Low computation overhead**: Due to the limited resources of IoT devices, the re-keying procedure should require as low computation resources on devices as possible.
- **Low communication overhead**: The re-keying procedure of the group should not induce a high number of transmitted messages in the network due to bandwidth limitations.
- **Low storage overhead**: The re-keying procedure should require low storage overhead on IoT devices.

One of the main challenges of the re-keying procedure is scalability [CS05]; in case a network consists of a large number of IoT devices, the groups of devices might be divided into subgroups. In such a case, a group controller is required within each subgroup to maintain and distribute the keys.

2.6.1 Group Key Management Approaches

Group key management schemes are classified into three categories [CS05]:

- **Centralized**: In centralized approaches, the key distribution function is assigned to a single entity that generates and distributes the keys whenever required. Centralized approaches are divided into three categories:
  - **Pairwise key:** In this subcategory, the key server shares a master secret key with each group member. These keys are utilized to establish a secure channel between the server and each member to distribute the session key securely whenever required.
  - **Broadcast/multicast secrets:** In this subcategory, the re-keying procedure of the members is based on broadcast or multicast messages instead of device-to-device separate transmissions.
  - **Hierarchy of keys:** A drawback of the pairwise approach is the requirement of sending the message to each individual member. To reduce the number of update messages, in this subcategory, the server shares secret keys with subgroups of members in addition to the individual channels. Whenever a member leaves a group, the server uses the secret of the subgroup that is not known by the leaving member

to distribute the new session key. As a result, the subgroup secret keys reduce the required number of transmitted messages.

- **Decentralized**: In decentralized approaches, a number of key manager servers are responsible for distributing the session group keys in order to provide scalability and avoid a single point of failure. This approach is divided into two subcategories [CS05]:
  - **Membership-driven re-keying:** Every time a member joins or leaves the group, the session group key is changed.
  - **Time-based re-keying:** In this subcategory, after a specific period of time, the session key is changed. Therefore, new members have to wait until the next re-keying to access the messages. Also, the members that have left will have access to the messages until the key times out.
- **Distributed**: In distributed approaches, the group members usually cooperate with each other to establish a group key. This is only applicable to networks where group members interact with each other. These approaches are out of the scope of this thesis.

2.7 Interoperability in IoT

Nowadays, different IoT vendors have various platforms, each with its own IoT infrastructure, protocols, incompatible standards, formats, and semantics. As a result, *IoT interoperability* has become demanding for different solutions to interact and exchange data or services. IEEE defines interoperability as "*the ability of two or more systems or components to exchange information and to use the information that has been exchanged* [Com+90]." In IoT systems, elements such as devices, services, and applications should be able to cooperate and communicate with each other. Interoperability was defined as a layered model with six layers by Tolk et al. [Tol04], which includes no connection, technical, syntactical, semantic, pragmatic/dynamic, and conceptual layers. Based on this layered model, different perspectives of interoperability were proposed in [NAG19] that include:

- **Device interoperability** concerns the exchange of information between heterogeneous devices.
- **Network interoperability** deals mainly with enabling mechanisms for message exchange between systems with different networks for end-to-end communication [NAG19]. Network interoperability should handle issues such as addressing, routing, resource optimization, and security [BZB17].
- **Syntactic interoperability** refers to the interoperation of the format and used data structures in exchanging information or services between IoT de-

vices. Syntactic rules are used by the sender and receiver to encode and decode messages, respectively.

- **Semantic interoperability** was defined by W3C as "*enabling different agents, services, and applications to exchange information, data and knowledge in a meaningful way, on and off the Web* [UM05]."
- **Platform interoperability** caused by the diverse platforms of IoT systems, including operating systems, programming languages, data structures, and architectures [NAG19]. Cross-platform interoperability can enable interoperability between different IoT platforms specific to a domain, such as smart home, smart healthcare, etc.

2.7.1 Interoperability Approaches

There are many approaches and mechanisms that can enable interoperability in IoT systems. We present an overview of these approaches in this section.

Physical Gateways

Physical gateways address interoperability using an intermediate tool between IoT systems, which act as a bridge between different specifications, standards, and networks. A one-to-one protocol gateway is a special hardware that enables interoperability between two different types of protocols (i.e., CoAP and MQTT). Gateways have scalability issues since different numbers of IoT devices interacting with each other are required to have specific connections to the gateway, which increase the time complexity of the system [NAG19]. For each one-to-one protocol translation, a separate gateway is required that imposes high costs on the system. In the layered network model, physical gateways are usually used to achieve interoperability in the network and transport layers.

For the application layer translation, middleware can be used, but middleware has limitations such as locking applications to a specific technology which further reduces interoperability. Protocol proxy is another alternative middleware for protocol interoperability in IoT systems. There are three proxies [DED17]: the forward proxy acts as the client, the reverse proxy acts as the server, and the interception proxy that acts as a man-in-the-middle. Among these three proxies, interception proxies can be used for application layer interoperability; however, proxies increase network delay since the translation is not on-demand, and all network traffic should be transmitted to the proxies. Similar to gateways, protocol proxies do not scale because of the complexity of the configuration and inefficiency of direct translation when the number of protocols increases [DED17].

Protocol Translators

Protocol translators are the replacement of hardware gateways or proxies for interoperability, and they are service-oriented architecture-based components [DED17]. In a service-oriented architecture, different services are used to enable distributed computing and remote system interaction. Protocol translators are active functions rather than being a passive network component. They can provide on-demand protocol translation without introducing any specific dependency to the applications in the system [DED17]. These translators can perform direct protocol to protocol translation and are intermediate in the system. Protocol translation in such translators should fulfill the following requirements [Der16]:

- **Transparency**: It should be transparent to the underlying network components and not add any extra configuration to the system by increasing the number of devices.
- **Scalability**: The translation should be easy to scale for hundreds or thousands of devices.
- **Security**: In the translation, distributed applications should be able to authenticate and authorize within the interoperability mechanism, and attacks such as man-in-the-middle attacks should not be possible.

Protocol translation can be done at different network layers, e.g., network, transport, and application layers. Lower layer protocol translation is more complicated than upper layer translation. This is primarily due to the complexity of these layers in handling packets, routing, forwarding, etc. However, if protocol translation is implemented at lower layers, it can be used in parallel with an upper layer translation. This thesis focuses on the protocol translation above the transport layer, and other lower layer translations are out of scope. (D)TLS is usually used to provide security for communications between application layer protocols. The security of protocol translation in these communications is usually guaranteed using *TLS interception*, which is explained below.

2.7.2 TLS Interception

In TLS, by default, an intruder cannot intercept the traffic between two parties. Meanwhile, there are some application use cases where network operators allow inspection, e.g., for malware detecting, content filtering, and network policy enforcement [EWD16]. This interception is legitimate, and the parties are informed. They agree on installing the certificate of a middlebox (issued by a Certificate Authority (CA)) in advance; the middlebox is typically a TLS proxy [JU12]. TLS interception is usually done through interception proxies, where a proxy is placed between client and server. Upon receiving a request from the client, it terminates the request and can analyze the actual plaintext. Then, the proxy makes another

request to the server on behalf of the client, and an end-to-end session is separated into two distinct but related sessions. The interception proxy must have access to the private key corresponding to the certificate it provides. For that, the interception proxy generates a new certificate and a key pair for the certificate, and they are used for a specific session. This certificate is signed by a CA the client and the server trust [JU12].

In TLS interception, the proxies can be deployed transparently or explicitly [WMY18]. In the explicit proxy, the client knows the IP address of the proxy, and a specific listening port is configured on the client. On the other hand, a transparent proxy can operate without the explicit awareness of the client. In such a case, the proxy intercepts outgoing requests that are sent to the server without the usage of any specific configuration on the client side. Nevertheless, for TLS interception, the certificate of the proxy must be added to the trusted root CA of the client certificate [WMY18].

2.8 Cloud Computing

Cloud computing is one of the most useful technologies that is widely used today to provide on-demand services and products. NIST [MG+11] defines cloud computing as "*a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction*". Over the last years, cloud computing has greatly impacted IoT networks [Bot+16]. Cloud computing and IoT networks are complementary technologies each with its own benefits and restrictions. In IoT, the devices have limited capabilities, and the network has scalability problems; on the other hand, cloud computing is scalable, and its resources are virtually unlimited. Thus, the integration of IoT and cloud computing has gained much attention [AAB15].

The main features of cloud computing are [Wan+10]:

- **On-demand resource provisioning**: Cloud computing provides resources and services on-demand, and users can customize their resources.
- **Scalability and flexibility**: Cloud-based resources are scalable and can scale up to additional resources as needed and later scale down. The cloud computing resources are flexible to adapt to a large number of end-users.
- **Guarantee QoS**: The resources provided by cloud computing guarantee QoS in terms of hardware performance, CPU speed, bandwidth, memory size, etc., for the users.
- **Automatic management**: Computing resources are managed transparently to users. Hardware and software-based resources in the clouds can be automatically reconfigured and orchestrated.

Virtualization and orchestration are among the primary enablers of cloud computing, and they are discussed in detail below.

2.8.1 Virtualized Environments

Virtualization systems are key enablers of cloud computing. Virtualization is defined as encapsulating the software layer from an operating system and providing the same behavior that is expected from physical hardware [PZH13]. In virtualized environments, applications can be decoupled from the underlying physical resources; these resources can be modified or transformed into various virtual or logical resources on-demand; this procedure is known as *provisioning*. The two most used virtualization technologies are [TRA15]:

- **Hardware virtualization** refers to abstracting the underlying hardware layers; the software performing this abstraction is a Hypervisor or a Virtual Machine Monitor (VMM). A hypervisor provides a virtual environment to the software that is equivalent to the host system, but it is decoupled from the state of the hardware. The hypervisor requires the provisioning of the operating system. The virtual environments are called Virtual Machines (VMs), in which different operating systems can be installed. Since VMs are independent of the state of the hardware, multiple VMs can run on the same hardware. Operating systems on the VMs are called *guest* operating systems.

- **System virtualization** is based on a lightweight virtualization process and can place an entire application with its dependencies inside a virtual *container* that is able to run on Linux distributions. Containers do not require provisioning of the operating system; meanwhile, they can also run inside a VM. A container is an abstraction at the application layer. It is a lightweight software package that packages dependencies like libraries, external third-party code packages, etc. [SKA18] that are required to execute the contained application. Different containers can run on the same machine in case they share the operating system's kernel with other containers.

The main difference between a container and a VM is that a VM virtualizes an entire machine from the operating system down to the hardware layers. However, a container only virtualizes the software layer above the operating system level. Containers require less disk and memory space compared to VMs, and they require less storage space [SKA18]. Containers also have better performance and scalability; they are easier to manage, and all of them are accessible from the host system [TRA15].

There exists a variety of applications for virtualization, such as the consolidation of physical servers, isolation of VMs or containers, debugging, etc. Setting up multiple physical systems to mitigate potential security risks or other purposes

have significant costs, including operational, physical, and technical costs. Physical machines require space, cabling, energy, administration, etc. Virtualization can help to decrease the above mentioned costs. Furthermore, it allows the installation of multiple VMs or containers on the same hardware and can protect systems from being a single point of failure [PZH13].

2.8.2 Container Orchestration

Orchestration refers to the control and management of different system architectures, including virtualized environments. Open Networking Foundation (ONF) defines Orchestration as "*the selection of resources to satisfy service demands in an optimal way, where the available resources, the service demands, and the optimization criteria are all subject to change*" [TR516]. Orchestration solutions are essential in managing virtualized environments and cloud data centers effectively.

Docker is an open-source platform that enables building, deploying, running, and managing containers. Container orchestration is used for system-level virtualization [TRA15], and it facilitates the deployment of applications among various Linux-based platforms. Container orchestration can automate the applications that are deployed inside the containers; it adds an extra layer of deployment engine on top of the containers. The container orchestration engine is used to monitor and control the containers, plan the actions to be performed, and coordinate different containers. It is also possible to monitor packaged applications and their status inside containers, perform updates, fix any errors, and restart them. One of the well-known container orchestration engines based on docker is Kubernetes.

Kubernetes is an open-source orchestration engine that was initially developed by Google. Kubernetes is a system-level virtualization orchestrator [TRA15]; it allows to scale both the computing resources available for the containers and the number of containers available to applications. Therefore, applications can scale up and down according to their needs. Besides the automatic scaling of containerized applications, Kubernetes can perform load balancing/spreading, disk management, etc. Kubernetes has some disadvantages, including the complexity of deploying and managing the containers [MK20].

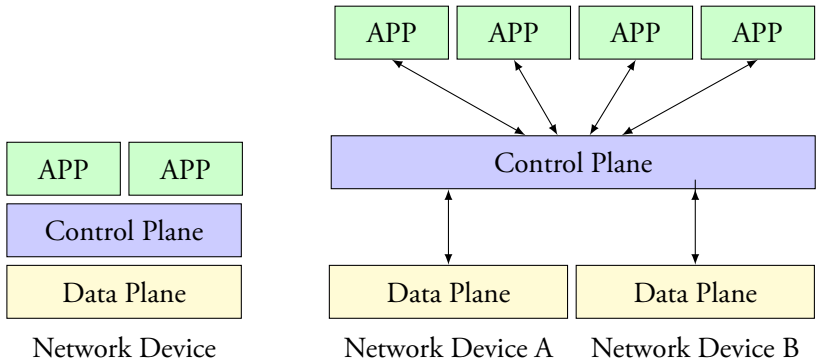
2.9 Software Defined Networking

The limitations of traditional networks with data traffic growth have become more evident. In traditional networks, each device has local control and data plane; such devices, e.g., switches and routers, are often vendor and application-specific, which makes their configuration and management complex [NPH20]. Low-level commands are required to configure each device separately [Kre+14]; other than configuration complexity, network environments have to adapt to load the changes in the network. The control and data plane on devices are responsible for policy definition and traffic forwarding, but they reduce flexibility and scalability.

Traditional networks cannot cope with the increasing demand and continuous expansion in the number of devices and applications brought through advances in cloud computing and IoT. Therefore, there is a shift from traditional networks toward Software-Defined Networking (SDN). SDN is an approach to enable network programmability, or increasing the network capacity to control network behavior dynamically via open interfaces [Hal+15]. SDN is designed to separate the network control or control plane from the forwarding process or data plane; this separation offers several benefits in network controllability and flexibility. It helps to implement a centralized network intelligence that makes network management and maintenance easy, and it enhances the network reactivity [BEE16]. In SDN networking, separating the control and data planes makes the network switches act as simple forwarding devices, and the control logic is implemented in a logically centralized controller that simplifies policy enforcement and network (re)configuration. A simplified view of the traditional network and SDN networking architecture is presented in Figure 2.4. In SDN, the routing and forwarding decisions of networking components (routers, and switches) are separated from the data plane. The network management becomes less complicated because the control plane only deals with the information regarding logical network topology, traffic routing, etc. In contrast, the data plane forwards the traffic related to the established configuration in the control plane.

The SDN architecture has four main characteristics [Kre+14]:

- The control and data planes are separated, and controlling functionality is removed from devices that will act as simple packet forwarding components.
- Forwarding decisions are flow specific instead of destination specific. A network flow is defined by a set of field values of a packet that can match a filter criterion or a set of actions. In the context of SDN, a flow is a number of packets between a specific source and a destination. Using the flow abstraction, the behavior of different network devices can be unified, and flow programming enables network flexibility that was limited to the capabilities of the devices' flow tables [McK+08b].
- Control logic is moved to a central entity called the SDN controller, which dictates the network policies. The SDN controller provides the required resources and abstractions to ease the programming of the data plane based on a logical and abstract view of the network. Its purpose is similar to a traditional operating system. There are many open-source platforms available for the controller such as Floodlight [Flo22], OpenDayLight [Ope22], and Ryu [Ryu22].
- The network is programmable with the help of the applications running on top of the SDN controller that communicates with the devices on the data plane. This is an essential characteristic of the SDN architecture, considering its main value proposition.



(a) Traditional network architecture **(b)** SDN architecture

Figure 2.4: Traditional network and SDN architecture

SDN has different interfaces, including northbound and southbound APIs. Northbound API is an interface between applications and the control plane, and it makes the information from the controller available to the applications. On the other hand, southbound API is an interface between the data and control planes. The southbound API allows control over the network and is used by the controller to dynamically re-configure and make changes to forwarding rules installed on the data plane devices [SNK12]. It also abstracts the information of physical network resources. As can be seen in Figure 2.4, the controller has direct control over the data plane elements via the southbound API, and the most common example of such an API is the OpenFlow [McK+08b] protocol. OpenFlow is described in more detail in the subsection below.

2.9.1 OpenFlow

OpenFlow [McK+08b] is a network protocol that enables managing traffic in a standardized way and specifies how a controller should communicate with other network components such as switches and routers. OpenFlow makes it possible to control the switches without requiring the vendors to reveal the code of their devices [LKR13]. OpenFlow has two logical components [BEE16]: a flow table that specifies how the packets need to be processed and forwarded in the network and an OpenFlow API that is responsible for handling the exchanges between switches, routers, and controllers. A flow table consists of a list of flow entries with matching fields and instructions. Incoming packets are checked against the match fields, and if a match is found, the packet is processed based on the action specified by that entry. OpenFlow makes it possible for software applications to program the flow table of switches in the network.

In the SDN architecture, the controller is responsible for controlling the flow tables of the switches with the help of the OpenFlow protocol. Using the Open-

Flow API, the controller can connect to the switches, manage them, receive packets from switches, and forward packets to switches [LKR13]. OpenFlow has several capabilities [LKR13] listed below:

- **Centralized control:** In OpenFlow, the controller has a wide knowledge of the network, and different switches are connected to a single controller. Therefore, the controller can make decisions for the broad part of the network.
- **Software-based traffic analysis:** This capability can be used to analyze network traffic in real-time. Since this analysis is performed by software, it is possible to use advanced features, including machine learning algorithms, for traffic analysis.
- **Dynamic update of forwarding rules:** In OpenFlow networks, forwarding rules in the flow table can be updated dynamically since the controller can modify flow table entries at any time without human interaction. The controller can use this feature to load balance the network traffic by dynamically changing the forward rules [LKR13].
- **Flow abstraction:** In OpenFlow, all traffic is abstracted as different flows in the flow table. For example, one flow can specify the whole TCP traffic; another flow can be all packets forwarding to the same destination IP address.

In OpenFlow, flow tables are installed on the OpenFlow switch. Open vSwitch (OVS) [Tu+21] is an open-source virtual OpenFlow switch that can work in virtualized environments; the OVS and its components are explained in more detail below.

2.9.2 Open vSwitch

OVS is a standard implementation of the OpenFlow protocol and is commonly used for OpenStack deployments. In OVS, packet forwarding is implemented on the datapath, and it consists of two datapaths [TSP17]: the slow path (ovs-vswitchd) handles the complexity of the OpenFlow protocol and the fast path (data path) that acts as a caching mechanism for packet lookup and delivery. OVS can be integrated with many operating systems, including Windows Hyper-V, Solaris, and Linux [TSP17]. OVS can work with hypervisors, e.g., KVM, and container systems such as docker, and it interconnects the virtual machines and containers. An overview of the OVS architecture and its components is depicted in Figure 2.5.

After the NIC inside the host OS receives the packets, OVS starts processing the packets in its datapath. A datapath can be deployed in the kernel or in userspace which requires additional firmware support. The datapath of the OVS

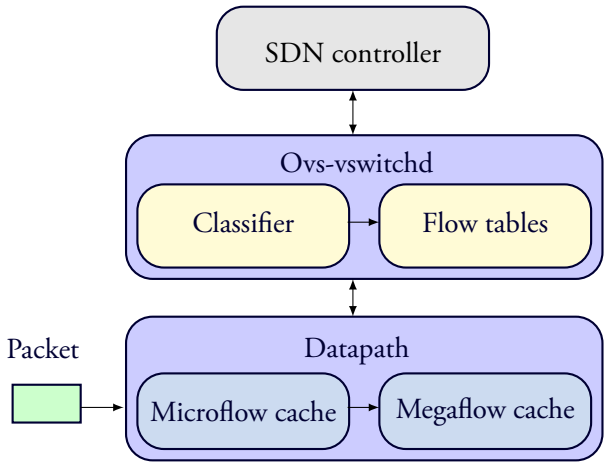


Figure 2.5: Open Vswitch components

first parses the packets to extract protocol headers that are required to perform the lookups. Then, it looks up the matching caches to determine which actions need to be performed on the packet (fast path). If no match is found in the caches, the packet is forwarded from datapath to the slow path or ovs-vswitchd by datapath, called *upcall*. In ovs-vswitchd the required information on processing and forwarding the packet is maintained. After a match is found in ovs-vswitchd flow tables, it passes the information regarding how to handle the packet to the datapath in the kernel or userspace.

As can be seen in Figure 2.5, two caches are used in the OVS datapath: microflow cache and megaflow cache [TSP17]. Microflow caches the entries that match the forwarding decision for connections at the transport layer, and the megaflow caches the forwarding decisions for aggregated traffic consist of different individual connections [TSP17]. In the datapath, the packet processing between these caches is as follows: if a matching entry is found in the microflow cache, the packet is later sent to the specific table in the megaflow cache to retrieve the required actions. In case of matching failure in the microflow, an expensive search is performed on every table in megaflow. If no match is found in both microflow and megaflow caches, an upcall is used to contact ovs-vswitchd. The ovs-vswitchd uses the flow classifier to find matching rules in the flow tables. Whenever a match is found, it will be forwarded to the datapath, and a new entry will be added to the relevant cache. Finally, the packet is forwarded to its destination. If ovs-vswitchd fails to find a match in its flow tables, it sends a packet-in request to the controller in the network to obtain information about the unknown packet.

Despite the flexibility offered by the SDN, OpenFlow enabled switches have fixed set of header fields [KKA21]. These headers can not be extended to meet the requirements of various applications. Moreover, SDN switches heavily de-

pend on the control plane to forward the packets, which increases the overhead of data-control communication [KKA21]. To resolve these issues, P4 (Programming Protocol-Independent Packet Processors) was designed. P4 is a programming language designed to change the packet forwarding behavior of the SDN switches. P4 expresses how packets are processed by a switch with the support of custom match/action tables and other forwarding related constructs. P4 has the capability to define the custom header formats and parse these headers dynamically. This language is protocol-independent, and if a new protocol is deployed in the network, it is easy to change the P4 program to support the new header fields. In this thesis, we only use SDN and OpenFlow, and P4 is not used.

2.10 Trusted Execution Environment

Confidentiality and integrity of sensitive code or data inside computing systems must be preserved against attacks over the network as well as attacks originating from software or hardware components on the same platform. This is because these systems can host multiple untrusted components. The problem of protecting sensitive data against co-located attackers raises the need for Trusted Execution Environments (TEEs) [Sch+22]. The demand to integrate trusted computing platforms into different systems, such as IoT networks, is also raised. TEE is a secure and isolated environment where sensitive data and code can be processed and run with higher levels of security than an operating system. A TEE provides an integrity-protected environment that consists of both memory and storage capabilities [Aso+14]. It can authenticate the executed code and guarantee the integrity of the runtime states, such as CPU registers, memory, and sensitive I/O. It can also preserve the confidentiality of running code, data, and runtime states stored on the persistent memory. The TEE content can be updated securely, and its content is not static. The TEE is resistant to software, and physical attacks performed on the main memory of the system. There are different solutions for TEE, such as Intel Software Guard Extensions (SGX), ARM TrustZone, and AMD Secure Encrypted Virtualization (SEV). In this thesis, we only use Intel SGX, which is explained below.

2.10.1 Intel SGX

Intel SGX [CD16] is a set of extensions to the Intel architecture that aims to provide integrity and confidentiality of sensitive data on a system that potentially has malicious components. SGX is able to isolate the execution environment from other applications and the operating system's kernel, as well as the hypervisor, with the help of *enclaves* [NBB20]. In SGX, enclaves are execution isolation environments that are able to run code and process sensitive data. Enclaves protect code and data from the outside enclave environment, including other applications on

the system and the operating system's kernel. In SGX, multiple enclaves can exist simultaneously; isolation is also provided between enclaves.

The execution process of SGX is as follows: a host process can start either in a normal or critical region. Normal code and data are loaded into the untrusted or normal region, while critical code and data are loaded into enclaves. External access to the enclaves is denied, and only the loaded code inside the enclave can access the data of Processor Reserved Memory (PRM). The result of the operation inside the enclave is later returned to the host process, while the enclave's code and data will remain protected in memory space.

Security features of SGX are listed below [Fei+21]:

- **Physical Memory Isolation**: The memory isolation feature is used to protect the enclave memory from unprivileged access during runtime. The PRM inside the enclave enables memory isolation protection at a hardware level; thus, any access to the isolated region is checked by the processor, and any unprivileged access is refused.

- **Data Sealing**: Data sealing can encrypt enclave data to store it persistently. Only the enclave that originally sealed data can decrypt it later.

- **Software Attestation**: SGX relies on software attestation [CD16] to prove that running code inside an enclave is trustworthy to the user and is hosted by trusted hardware.

SGX is vulnerable to various side channel attacks [NBB20] that can be used to break the confidentiality of enclaves. However, there are some techniques, such as applying microcode patches or changing application design, that can be utilized to mitigate such attacks.

2.10.2 Attestation

To check whether an enclave code has been appropriately instantiated and the data is trustworthy, SGX uses two attestation mechanisms [Fei+21]: local attestation and remote attestation. In this thesis, we only use the local attestation. The local attestation is performed between two enclaves running on the same platform to authenticate each other. The procedure of local attestation is depicted in Figure 2.6. In this attestation, a communication channel is established between a prover enclave and a verifier enclave. The prover enclave obtains the measurement of the other enclave or verifier enclave [Fei+21]. The measurement is used to establish the identity of verifier enclave. The prover enclave generates a REPORT structure and then forwards it to the verifier. After receiving the REPORT structure by the verifier, it retrieves the Report Key from the REPORT structure through the EGETKEY instruction. Subsequently, the verifier uses the Report Key to generate the MAC value over the REPORT structure and verifies whether the REPORT is

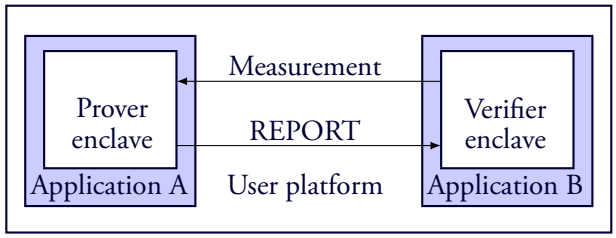


Figure 2.6: SGX local attestation

authenticated or not. If this verification succeeds, the verifier can determine that the prover enclave is trustworthy.

Remote attestation is an extended version of the local attestation; in that, an enclave can prove its identity to a remote party (challenger) [Fei+21]. The goal of remote attestation is to enable a remote party to determine the level of trust in another system. Using remote attestation, an enclave can present evidence to a remote party. Based on that evidence, the remote party can decide on authorization decisions. The remote attestation procedure is depicted in Figure 2.7. As can be seen, remote attestation uses a special enclave or quoting enclave to verify the REPORT and transforms it into a structure called a QUOTE structure. First, a communication channel is established between a sample application and a challenger that can be an attestation service provider. Then, the challenger issues a challenge to the application to check whether the enclave and the platform are legitimate. The application sends the quoting enclave's identity with the challenge to the application enclave. The application enclave generates a response to the challenge that also contains an ephemeral public key. The application enclave then generates and sends the REPORT structure to the application. Then the REPORT structure is sent by the application to the quoting enclave for signing [Fei+21]. The quoting enclave first extracts the Report Key and verifies the MAC of the REPORT. The quoting enclave creates the QUOTE structure, signs it, and then sends it to the application. The application sends the QUOTE structure to the server, and the server verifies the QUOTE signature. The server also checks the integrity of the QUOTE.

2.10.3 Other Trusted Execution Environments

Intel SGX is one of the solutions to provide a trusted execution environment; there are other solutions, including Intel Trust Domain Extensions (Intel TDX). TDX is a newer architecture than SGX, and it deploys hardware-isolated virtual machines called trusted domains (TDs). In TDX, the virtual machines are isolated from VMM, hypervisor, or other non-TD software on the platform. TDX can provide memory and CPU state confidentiality and integrity to keep sensitive data secure from various attacks. TDX supports remote attestation, in compared to SGX,

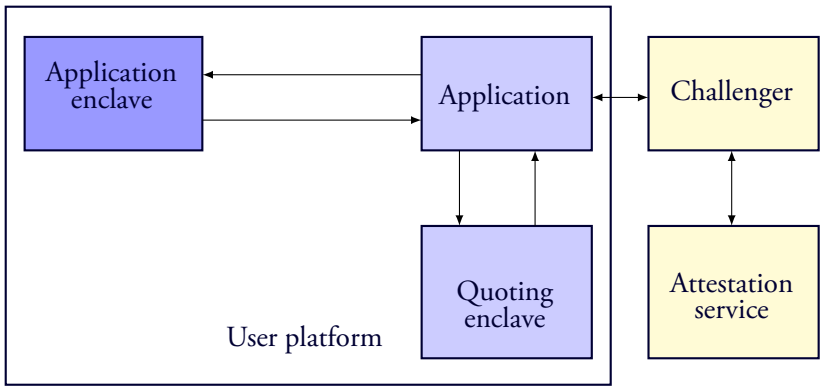


Figure 2.7: SGX remote attestation

TDX attestation provides more evidence and measurements from TD [SMF21].

Another trusted execution environment is ARM TrustZone, which can also provide an isolated execution environment. It includes security extensions to ARM System-On-Chip (SoC) that cover the processor, memory, and peripherals [Hua+17]. TrustZone provides hardware-based access control of hardware resources by enabling a processor to run in two execution environments. TrustZone splits the processor into a normal world and a secure world. Each world has its own user space and kernel space, cache, and other resources. The normal world cannot get access to the secure world's resources, while the secure world can access all of the resources [Hua+17].

AMD SEV is another TEE that aims to provide confidential computing for virtual machines or containers [ABT16]. It is a hardware-assisted TEE mainly responsible for encryption and protecting the system memory of different cloud infrastructures. AMD SEV enables encryption of the main memory by encrypting each individual VM memory space with a unique secret key associated with each guest VM and protecting the key from other VM attacks, or malicious hypervisors [ABT16]. AMD SEV uses Secure Memory Encryption (SME) to protect the main memory against physical attacks. The problem of the original SEV is that it could leak sensitive information during interrupts from guests to the hypervisor through registers [Mén+22; HB17; Wer+19], and this problem was addressed in the extended version of SEV or SEV Encrypted State (SEV-ES). In SEV-ES, the register states are encrypted using the guest-specific encryption key, and the guest operating system needs to give access of the guest registers to the hypervisor.

2.11 Formal Verification

Security protocols are used in different areas, such as e-commerce, wireless networks, and IoT networks, and the design of these protocols is error-prone [Bla+16].

Some security errors cannot be found by functional testing; hence, *formal verification* is desirable. Formal verification helps in proving the correctness of the protocols with the help of a formal mathematical model of the protocol. Two main models used to verify protocols are [Bla+16]:

- In the **Symbolic model** or Dolev-Yao model cryptographic primitives are modeled by function symbols in an algebra of terms with different equations. In this model, messages are terms of these primitives.
- In the **Computational model** messages are represented as bitstrings, and cryptographic primitives are functions from one bitstring to another bitstring.

The symbolic model is easier to build than the computational model, and it is used in many automatic verification tools such as AVISPA [Arm+05], Tamarin [Mei+13], and ProVerif [Bla+18]. However, the computational model is closer to the real execution of the protocols, and it is more challenging to automate [Bla+16]. In this thesis (papers III, V, and VI), we used ProVerif as an automatic verifier tool.

The basic security properties required by most protocols are *secrecy* and *authentication* [Bla+16]. Secrecy means that the adversary cannot gain information on data transferred and manipulated by the protocol. The secrecy property is formalized in two ways: 1) *secrecy* which means the adversary cannot compute the actual data, and 2) *strong secrecy* means that the adversary cannot detect any change in the secret value (or has no information about the secret value at all). In order to verify the protocols, a set of terms (messages) that an adversary knows is computed, and if a message does not belong to this set, the secrecy of the message is preserved.

Authentication means that if participant A runs the protocol with participant B, then B runs the protocol with A [Bla+16]. Correspondence properties are used to formalize authentication. Correspondence indicates that if A executes an event e_1, then B has executed event e_2 previously. Correspondence has two different variants including *injective correspondence* and *non-injective correspondence*. Injective correspondence means that each execution of e_1 is corresponding to a distinct execution of e_2, but if e_2 has been executed at least once, this is referred to as non-injective correspondence.

2.11.1 ProVerif Modeling

As mentioned earlier, ProVerif is a symbolic protocol verifier representing protocols with Horn clauses. Horn clauses are formulas in the form of $F_1 \wedge ... \wedge F_N$ where different Fs are facts [Bla+16]. ProVerif takes as input a protocol that was modeled with applied pi calculus (a language for describing and analyzing security protocols) and the security properties that we want to prove. Then, ProVerif automatically translates the model into Horn clauses and a set of security properties.

Then ProVerif determines whether a fact is derivable from the clauses or not. If it is derivable, then the security property is proved.

The modeling starts by defining terms, expressions, and processes. Terms can have different types, such as boolean, bitstring, or other user-defined types. Constructors are represented with function symbols. They are used to make terms modeling primitives used in cryptographic protocols such as hash functions, encryptions, and digital signatures. Destructors are used to manipulate terms formed by constructors. As an example, the asymmetric encryption is represented with a constructor as:

```
aenc(bitstring, pkey): bitstring
```

where *pkey* is the type of public key. The corresponding destructor is defined as:

```
adec(aenc(x, pk(y)), y) → x
```

It can decrypt the cipher text *aenc(x, pk(y))* where *y* is the secret key and *pk(y)* is the public key.

2.11.2 Verifying Security Properties

In ProVerif, the adversary is capable of intercepting, computing, and sending all messages; in other words, it has the Dolev-Yao adversary model. In the modeling, a process P preserves the secrecy of message M if it cannot be read on a public channel in the process run by any adversary [Bla+18]. To test if the adversary can gain access to message M, we use the following query:

```
query attacker(M)
```

where M is a term that is private and not known to the attacker.

As mentioned earlier, correspondence indicates the relationship between events. An event is defined as *e(*$t_1, ..., t_n$*)* where $t_1, ..., t_n$ declare the types of the event arguments. The correspondence assertion is defined as:

```
query x1 : t1 , ... , xn : tn ;
event(e(M1, ..., Mj)) ⇒ event(e'(N1, ..., Nk))
```

where different Ms and Ns are the terms built by constructors and xs are variables of types $t_1, ..., t_n$. The above query shows a non-injective correspondence and it is satisfied if for each occurrence of *e(*$M_1, ..., M_j$*)* there is a previous occurrence of *e'(*$N_1, ..., N_k$*)*.

Injective correspondence shows a one-to-one relation between events. It is desirable in cases such as financial transactions where a server should complete the payment transaction only once for each transaction started by a client [Bla+18]. The injective correspondence is denoted as follows:

```
query x1 : t1 , ... , xn : tn ;
inj-event(e(M1, ..., Mj)) ⇒ inj-event(e'(N1, ..., Nk))
```

In papers III, V, and VI, we used ProVerif modeling to model the protocols and verify security properties. In verifying security properties, we use both injective and non-injective correspondence.

In this chapter, we present the various research background topics used in this thesis. There are many security aspects and challenges in deploying software updates, especially in IoT devices with limited resources. The main aspects covered in this thesis are handling vulnerabilities, deploying actual software updates, translating protocols, and managing secret keys. In the next chapter, we demonstrate our contributions to each paper and conclude with the final remarks.

Contributions and Conclusions

This thesis aims to identify and design solutions that can make secure and lightweight updates feasible in IoT environments. The different contributions of this thesis are visualized within the areas as presented in Figure 3.1. These contributions are described in more detail in section 3.1, followed by the conclusions in section 3.2.

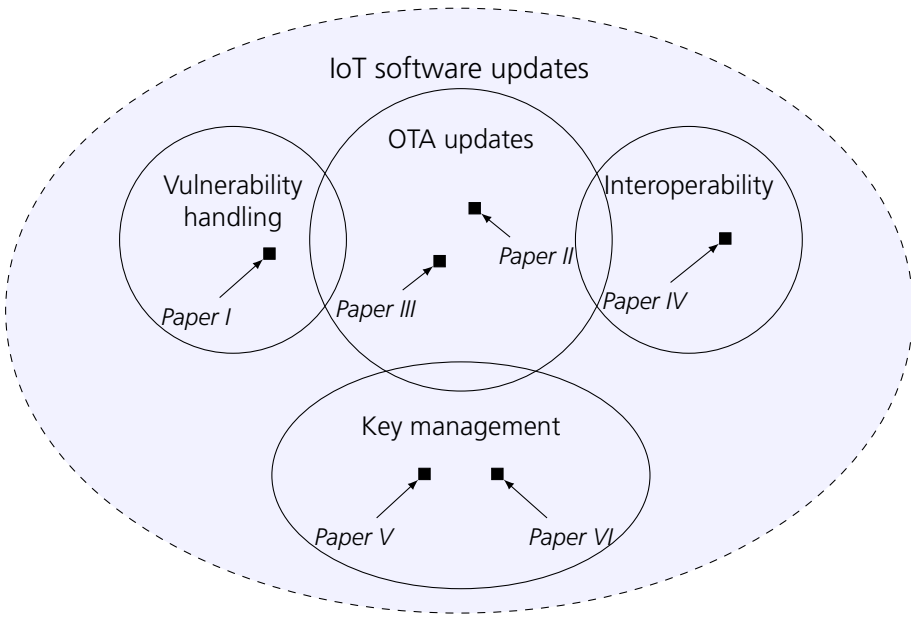


Figure 3.1: The different contributions of this thesis. Each paper is located in the area it contributes mostly.

3.1 Contributions

This thesis contains six papers and each paper contribution is described below.

3.1.1 HAVOSS: A Maturity Model for Handling Vulnerabilities in Third Party OSS Components

In paper I, we present and evaluate a maturity model (HAVOSS) for managing vulnerabilities in third-party OSS and COTS components. Software maintenance is an important aspect of the software development life cycle, and it mainly focuses on changing software to include new functionalities or fixing errors and bugs. Due to the high increase in the number of attacks on software systems in recent years, identifying and handling security vulnerabilities is an important aspect of software maintenance. The usage of open-source software (OSS components) can decrease development and maintenance time compared to developing all the code in-house. However, in the case of using open-source components, handling urgent updates is difficult and costly, and a mechanism is required to handle new vulnerabilities efficiently.

A maturity model can be viewed as a tool that can help organizations to describe how well their practices and processes can generate the desired outcomes. It helps organizations to improve their way of working by introducing the required changes. In paper I, we design a maturity model called HAVOSS (HAndling Vulnerabilities in third-party OSS), and its main focus is on identifying and managing vulnerabilities in third-party code. HAVOSS is built upon other existing models for secure software maintenance, such as CMMI, BSIMM, SAMM, etc., and the published guidelines for the security of IoT devices. It is not a replacement for other models; it is mostly intended to complement other maturity models. HAVOSS consists of 6 capability areas and 21 practices in total; these capability areas are product knowledge, identification and monitoring of sources, evaluating vulnerabilities, the remedy of vulnerabilities, and delivering updates and communication.

We take the following steps in designing HAVOSS; based on the identified need, a first version of the model was designed. For that, the responses of interviews from practitioners on how they handle their vulnerabilities and other relevant maturity models were used. Then, the first version of the model was iteratively improved using two offline evaluation rounds. In the first round of evaluation, we refined the model based on feedback from practitioners. In the second round, we focused on the importance of the practices inside the model and asked practitioners to rate the importance of practices. The evaluations indicate that our model practices are seen as important by the practitioners.

3.1.2 Energy Consumption for Securing Lightweight IoT Protocols

In paper II, we compare the energy consumption of CoAP and MQTT protocols in performing secure OTA updates. We also analyze the energy consumption of using TLS and DTLS in these two protocols with different security options. CoAP and MQTT are the two most common application-level protocols in IoT. CoAP is a lightweight request and response protocol that runs over UDP. On the other hand, MQTT (runs over TCP) is a publish/subscribe protocol designed for low bandwidth environments. DTLS and TLS are used for security at the transport layer in CoAP and MQTT, respectively.

For measuring energy consumption, we consider a use case scenario in which a sensor (IoT device) sends data to a server for analysis. The sensor uses both in-house developed and third-party codes and is subject to different vulnerabilities that requires a regular update. In this work, we compare the cost of adding security to CoAP and MQTT, the difference between various modes of operations in AES, and the effect of different key sizes. Furthermore, we analyze the cost of handshake using different cipher suites and the cost of updating the device firmware securely.

We perform the tests on ESP32, a popular development board for IoT, and the result of sending encrypted data indicates that MQTT in larger payload sizes has an advantage over CoAP since, in CoAP, the payload size is bound to 1024 bytes. Regarding the handshake, we conclude that ECDHE should always be preferred to DHE. Due to the asymmetric cost of signing and verification in RSA, it can be used if client authentication is not needed. In firmware updates, MQTT is again more efficient (a factor of 4 for energy and a factor of 2 for time) than CoAP.

3.1.3 RoSym: Robust Symmetric Key Based IoT Software Upgrade Over-the-Air

In paper III, we propose RoSym, a secure and symmetric-based upgrade scheme for IoT environments. IoT is a system of interconnected heterogeneous devices. IoT devices are usually resource-constrained with limited resources, including battery, memory, power, etc. IoT devices are mostly controlled remotely, and software updates or patches are performed Over-The-Air (OTA). Since IoT devices are battery-powered with limited resources, they are an attractive target of DoS attacks over the network. This motivates using a secure and robust OTA upgrade mechanism in IoT. However, existing upgrade solutions are primarily based on heavy public key operations and were designed for Wireless Sensor Networks (WSNs). These solutions do not cope with the characteristics of IoT networks, and they do not provide quantum resistance.

A secure and robust software upgrade scheme for resource constrained devices should have integrity and confidentiality protection, DoS attack protection, efficient communication and low computation overhead, and low memory requirements. Considering these requirements and the issues of other existing schemes, we designed the RoSym upgrade scheme. RoSym is a lightweight and symmetric

key based scheme, and in that larger key sizes can be used to resist quantum search algorithms. RoSym is robust, and it can protect against battery drain DoS attacks. In RoSym, a central unit called Device Management System (DMS) is responsible for upgrade preparations and the upgrade itself. To protect against a single point of failure, several DMSs can be used.

The software upgrade or patch is divided into different packages, and confidentiality of packages is provided using a symmetric key. In order to protect the integrity of these packages, we use MAC verification, and for that, an integrity protection key is used. The encryption and integrity protection keys are transferred to the devices using a secure channel, and these keys will be revoked at the end of a successful upgrade procedure. To protect against DoS attacks, other than MAC verification, we use two time thresholds; one threshold defines the maximum delay between individual package arrival times, and the other threshold defines the total allowed upgrade time. Whenever the packages are received on the IoT device, the local arrival times of the packages are captured by the devices. The difference between different packages' arrival times is checked against the thresholds. If the time difference exceeds the allowed thresholds, the upgrade will be aborted.

RoSym is implemented and tested on a real testbed setup using ESP32-S2 boards, and the result of our evaluation indicate that RoSym is efficient with respect to computation, communication, and memory overhead. We also use formal verification using ProVerif to prove the security features of RoSym, including confidentiality, integrity, and DoS resistance, and ProVerif verifies these features using secrecy and correspondence assertions.

3.1.4 Chuchotage: In-line Software Network Protocol Translator for (D)TLS

In paper IV, we design Chuchotage, an on-demand, in-line TLS interception and confidential solution to provide interoperability for IoT. Due to the heterogeneous nature of IoT devices, they use many incompatible protocols to communicate with each other or to communicate with a back-end server. This communication is enabled through protocol translation, using hardware translators or gateways; however, these solutions cannot address security, latency, and scalability requirements, especially in IoT environments. To address these requirements and enable a secure massive M2M communication, we design Chuchotage.

Chuchotage uses prior technologies, including Software Defined Networking, Trusted Execution Environments, and TLS interception. The translation is performed on-demand with the help of translator boxes created by software switches on the network path. This translation is done at the application layer, and we target the application layer since it has a great impact on application performance. During translation, secure TLS interception is provided using TEEs, and TEEs provide confidentiality and integrity with the help of isolated execution environ-

ments. TEEs are used on the network path to decrypt, translate, and re-encrypt data.

SDN decouples data and control planes and in that controller has a global view over the network. OpenFlow protocol is usually used in SDN and enables the communication between the controller and other network components. OpenFlow can be used with both hardware and software switches, and in Chuchotage, we used the software switch Open vSwitch. Open vSwitch is used to translate network flows between endpoints. Whenever a translation action is triggered by the switch, a translator box is created inside a TEE. Translator boxes can be child processes of the switch, or they can be external. External translator boxes are instantiated by the controller. For TEE, we use Intel SGX in our implementation, but other TEEs can be used as well. After the creation of a translator box, its trustworthiness is attested by a verifier network function, then the required cryptographic materials for TLS interception are provisioned to the TEE. The translator box creation is network-flow specific, and different translator boxes will be created for distinct flows; this solves scalability issues. For TLS interception inside the translator boxes, we use the ME-TLS protocol. ME-TLS allows cryptographic key materials to be delivered to the translator boxes in-band, does not require additional TLS connections or round-trips, and is compatible with TLS 1.3.

For the evaluation of Chuchotage, we translate two popular IoT protocols, secure CoAP and HTTP, and we use Ryu as the SDN controller. We evaluate the solution with several tests, including sending packet batches with different sizes or sending packet batches with different payload sizes. We measure translation and transmission times with and without using the TEE (Intel SGX in our case). We conclude that with the use of TEE, there is a slight increase in the translation and transmission time, but this increase depends to the choice of the TEE.

3.1.5 LMGROUP: A Lightweight Multicast Group Key Management for IoT Networks

In paper V, we design LMGROUP, which is a lightweight and multicast group key establishment protocol for IoT networks; it is based on the Elliptic Curve Integrated Encryption Scheme (ECIES) and HMAC verification. Due to the limitations of IoT devices (limited resources) and limitations of IoT networks (low bandwidth), multicast group communication is preferred. Multicast communication is especially useful when software updates are required to be sent to a large number of devices. To enable a secure multicast group communication, a group key needs to be established to the group members. There are a variety of group key establishment protocols, mainly designed for WSN. In WSN, sensors have high interaction with each other to share the group key, but this is usually not the case in most IoT networks, e.g., when the device sends periodic aggregated data to the server for further analysis. The usage of multicast transfer instead of broadcast-

ing is also preferred due to availability reasons, since in broadcasting, in case of unexpected errors, the availability of all devices can be endangered.

For multicast group communication, a central server (resource-rich node) needs to decide the group members; this can be done by the network administrator or by using automatic member selection algorithms. Automatic algorithms are preferred to make the group members less predictable to the attacker. After selecting the group members, the group key needs to be established, and this key needs to be renewed frequently due to changes in the group members to provide forward secrecy. This change can be done for many reasons, including physical maintenance, adding new members, etc. For the group key establishment, we design LMGROUP, a lightweight multicast group key management scheme for small to large-sized networks. In LMGROUP, the devices do not interact in any way to establish the group key, and the devices cannot decide which group they want to belong to, and these are only performed by the server(s).

Our group member selection algorithm is based on the criticality levels of the IoT devices in the network (how critical is the role of an IoT device), and it makes sure not all critical devices are grouped together, e.g., the main entrance doors of a hospital. After deciding about the group members, a hint is piggybacked along with the acknowledgment of the previously received message to the group members. This hint carries information about the new group and can be used as the seed of a key derivation function to extract an authentication key which is used later to authenticate group key establishment messages in HMAC verification. The actual key establishment phase is based on ECIES operations, and after receiving messages, the group members can authenticate the messages and calculate the group key.

We implement LMGROUP on a real testbed setup using ESP32-S2, and the results indicate that it is lightweight in case of communication, computation, and memory overhead. We also indicate that LMGROUP is scalable for large group sizes. We use formal verification to formally verify LMGROUP, and we verify that it is secure against different attacks, including MITM, replay, impersonation, and DoS attacks.

3.1.6 Flowrider: Fast On-Demand Key Provisioning for Cloud Networks

In paper VI, we present Flowrider, a key provisioning mechanism for cloud networks that uses symmetric keys and can significantly reduce the related computational load on network endpoints. Cloud computing gained more attention in recent decades and has many advantages, including unlimited resources and scalability. Cloud computing can also be integrated with IoT to provide these advantages for IoT. SDN is one of the key enabling technologies of cloud computing; however, network security operations did not cope with the capabilities of SDN, and public key cryptography rather than symmetric key encryption is

the main choice in such networks. Public key cryptography is CPU-expensive, and it has key management complexity. Endpoints may even lack sufficient computational resources to generate public key material. Generating symmetric keys requires less computational power but secure key provisioning and authentication are more complex than public key encryption. In Flowrider, we use the capabilities of SDN networking to provision key materials, and it embeds key distribution into the flow establishment of the network. In Flowrider, key distribution is agnostic to the network topology or architecture. Flowrider enables flow-specific symmetric key in such a way that each flow is isolated from other network traffic and is compatible with (D)TLS v1.2 and v1.3.

Whenever a client (device) wants to establish a secure session with a server, it can use either symmetric pre-shared keys or public key-based material. However, if the client and server use multiple flows, communications on each network flow will be secured with the same key material. Compromising key materials of one flow can endanger the security of all flows. Therefore enabling flow-specific key provisioning is an advantage that can be gained using Flowrider. In Flowrider, each time a client initiates a session with the server and triggers flow establishment, the SDN network controller generates a new pre-shared key for that flow and provisions it to both endpoints through the pre-established secure channel. In case of a compromise, the controller can revoke the flow keys issued to the endpoints.

We implement and test Flowrider using the Ryu SDN controller. The results of the experiments indicate that Flowrider performs much better in the case of task clock and CPU utilization compared to establishing public key-based material. We also successfully verify the security properties of Flowrider, including the secure provisioning and secrecy of pre-shared keys associated with the flow and mutual secure possession of keys by the endpoints using formal verification with ProVerif.

3.2 Conclusions

The main focus of this dissertation is to analyze and design different security mechanisms for deploying secure software updates in IoT. Our contributions cover several aspects, including vulnerability handling, OTA updates, interoperability, and key management.

The diversity of the contributions indicates that in a secure software update, especially in IoT, there are many security aspects that cannot be neglected. Failing to provide security in any aspect can result in compromising the security of the whole update procedure. In a secure software update, vulnerabilities need to be identified and handled, and a secure OTA mechanism based on the characteristics of the network needs to be selected or designed. In case of communication between incompatible protocols, an efficient and secure solution to enable interoperability need to be used. Furthermore, a lightweight key management scheme must be used to renew the secret keys.

The secure software update is an essential part of software maintenance; before the actual update can take place, vulnerabilities need to be identified. Our maturity model presented in paper I, can be used for efficient vulnerability identification and handling. After that, a secure update mechanism needs to be utilized to transfer updates or patches to the endpoints; in paper II, we compare two common IoT protocols (CoAP and MQTT) and find out which one is more energy efficient for specific use cases. Later in paper III, we design a lightweight scheme for OTA upgrade in IoT, considering the requirements of IoT networks and devices. Due to the heterogeneous nature of IoT, devices should be able to communicate securely with incompatible protocols. To make this possible, in paper IV, we design a solution that can be used to translate protocols to each other securely. The secret keys need to be renewed frequently due to any changes in the network, and in papers V and VI, we present mechanisms that can be used for key management and key provisioning. Finally, all of these pieces together can make the secure software updates possible. To deliver secure updates, developers and software maintainers need to pay special attention to all of these aspects.

References

[AAB15] Z. H. Ali, H. A. Ali, and M. M. Badawy. "Internet of Things (IoT): definitions, challenges and recent research directions". In: *International Journal of Computer Applications* 128.1 (2015), pp. 37–47.

[ABT16] O. AbdElRahem, A. M. Bahaa-Eldin, and A. Taha. "Virtualization security: A survey". In: *2016 11th International Conference on Computer Engineering & Systems (ICCES)*. IEEE. 2016, pp. 32–40.

[Arm+05] A. Armando et al. "The AVISPA tool for the automated validation of internet security protocols and applications". In: *International conference on computer aided verification*. Springer. 2005, pp. 281–285.

[Asc+12] N. Aschenbruck et al. "Selective and secure over-the-air programming for wireless sensor networks". In: *2012 21st International Conference on Computer Communications and Networks (ICCCN)*. IEEE. 2012, pp. 1–6.

[Aso+14] N. Asokan et al. "Mobile trusted computing". In: *Proceedings of the IEEE* 102.8 (2014), pp. 1189–1206.

[BCS12] C. Bormann, A. P. Castellani, and Z. Shelby. "Coap: An application protocol for billions of tiny internet nodes". In: *IEEE Internet Computing* 16.2 (2012), pp. 62–67.

[Bec+10] J. Becker et al. "Maturity models in IS research". In: (2010).

[BEE16] K. Benzekki, A. El Fergougui, and A. Elbelrhiti Elalaoui. "Software-defined networking (SDN): a survey". In: *Security and communication networks* 9.18 (2016), pp. 5803–5833.

[Ben+97] C. H. Bennett et al. "Strengths and weaknesses of quantum computing". In: *SIAM journal on Computing* 26.5 (1997), pp. 1510–1523.

[Bjo09] K. Bjoernsen. “Koblitz Curves and its practical uses in Bitcoin security”. In: *order (ε (GF (2k)* 2.1 (2009), p. 7.

[BJS07] E. B. Barker, D. Johnson, and M. E. Smid. *SP 800-56A. recommendation for pair-wise key establishment schemes using discrete logarithm cryptography (revised)*. 2007.

[BKP09] J. Becker, R. Knackstedt, and J. Pöppelbuß. “Developing maturity models for IT management”. In: *Business & Information Systems Engineering* 1.3 (2009), pp. 213–222.

[Bla+16] B. Blanchet et al. “Modeling and verifying security protocols with the applied pi calculus and ProVerif”. In: *Foundations and Trends® in Privacy and Security* 1.1-2 (2016), pp. 1–135.

[Bla+18] B. Blanchet et al. “ProVerif 2.00: automatic cryptographic protocol verifier, user manual and tutorial”. In: *Version from* (2018), pp. 05–16.

[Blu+92] C. Blundo et al. “Perfectly-secure key distribution for dynamic conferences”. In: *Annual international cryptology conference*. Springer. 1992, pp. 471–486.

[Bot+16] A. Botta et al. “Integration of cloud computing and internet of things: a survey”. In: *Future generation computer systems* 56 (2016), pp. 684–700.

[BWP16] D. Badampudi, C. Wohlin, and K. Petersen. “Software component decision-making: In-house, OSS, COTS or outsourcing-A systematic literature review”. In: *Journal of Systems and Software* 121 (2016), pp. 105–124.

[BZB17] O. Bello, S. Zeadally, and M. Badra. “Network layer inter-operation of Device-to-Device communication technologies in Internet of Things (IoT)”. In: *Ad Hoc Networks* 57 (2017), pp. 52–62.

[CD16] V. Costan and S. Devadas. “Intel SGX explained”. In: *Cryptology ePrint Archive* (2016).

[Che+16] L. Chen et al. *Report on post-quantum cryptography*. Vol. 12. US Department of Commerce, National Institute of Standards and Technology …, 2016.

[Com+90] I. S. C. Committee et al. “IEEE standard glossary of software engineering terminology (IEEE Std 610.12-1990). Los Alamitos”. In: *CA: IEEE Computer Society* 169 (1990), p. 132.

[CS05] Y. Challal and H. Seba. “Group key management protocols: A novel taxonomy”. In: *International J. Inf. Technol.* Citeseer. 2005.

[DCD17] S. Debnath, A. Chattopadhyay, and S. Dutta. “Brief review on journey of secured hash algorithms”. In: *2017 4th International Conference on Opto-Electronics and Applied Optics (Optronix)*. IEEE. 2017, pp. 1–5.

[De +05] T. De Bruin et al. “Understanding the main phases of developing a maturity assessment model”. In: *Australasian Conference on Information Systems (ACIS)*. Australasian Chapter of the Association for Information Systems. 2005, pp. 8–19.

[DED17] H. Derhamy, J. Eliasson, and J. Delsing. “IoT interoperability—on-demand and low latency transparent multiprotocol translator”. In: *IEEE Internet of Things Journal* 4.5 (2017), pp. 1754–1763.

[Der16] H. Derhamy. “Towards Interoperable Industrial Internet of Things: An On-Demand Multi-Protocol Translator Service”. PhD thesis. 2016.

[DR08a] T. Dierks and E. Rescorla. “The transport layer security (TLS) protocol version 1.2”. In: (2008).

[Dwo+01] M. J. Dworkin et al. “Advanced encryption standard (AES)”. In: (2001).

[Dwo+15] M. J. Dworkin et al. “SHA-3 standard: Permutation-based hash and extendable-output functions”. In: (2015).

[EB22] S. El Jaouhari and E. Bouvet. “Secure firmware Over-The-Air updates for IoT: Survey, challenges, and discussions”. In: *Internet of Things* 18 (2022), p. 100508.

[Elh+18] S. Elhadi et al. “Comparative study of IoT protocols”. In: *Smart Application and Data Analysis for Smart Cities (SADASC’18)* (2018).

[ET05a] P. Eronen and H. Tschofenig. *Pre-shared key ciphersuites for transport layer security (TLS)*. Tech. rep. RFC 4279, December, 2005.

[EWD16] F. Erlacher, S. Woertz, and F. Dressler. “A TLS interception proxy with real-time libpcap export”. In: *41st IEEE Conference on Local Computer Networks (LCN 2016), Demo Session*. 2016.

[Fei+21] S. Fei et al. “Security vulnerabilities of SGX and countermeasures: A survey”. In: *ACM Computing Surveys (CSUR)* 54.6 (2021), pp. 1–36.

[Flo22] Floodlight. *Floodli ght Controller*. `https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview`. Accessed: 2022-07-11. 2022.

[FRP17] D. Frisch, S. Reißmann, and C. Pape. "An over the air update mechanism for ESP8266 microcontrollers". In: *Proceedings of the ICSNC, the Twelfth International Conference on Systems and Networks Communications, Athens, Greece*. 2017, pp. 8–12.

[Gro96a] L. K. Grover. "A fast quantum mechanical algorithm for database search". In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 1996, pp. 212–219.

[GS15] A. Gupta and S. Sharma. "Software maintenance: Challenges and issues". In: *Issues* 1.1 (2015), pp. 23–25.

[Gun+21] M. Gunnarsson et al. "Evaluating the performance of the OSCORE security protocol in constrained IoT environments". In: *Internet of Things* 13 (2021), p. 100333.

[Hal+15] E. Haleplidis et al. *Software-defined networking (SDN): Layers and architecture terminology*. Tech. rep. 2015.

[HB17] F. Hetzelt and R. Buhren. "Security analysis of encrypted virtual machines". In: *ACM SIGPLAN Notices* 52.7 (2017), pp. 129–142.

[Hee+11] T. Heer et al. "Security Challenges in the IP-based Internet of Things". In: *Wireless Personal Communications* 61.3 (2011), pp. 527–542.

[Hel+76] M. Hellman et al. "New directions in cryptography". In: (1976).

[HGC20] S. Halder, A. Ghosal, and M. Conti. "Secure over-the-air software updates in connected vehicles: A survey". In: *Computer Networks* 178 (2020), p. 107343.

[Hua+17] Z. Hua et al. "{vTZ}: Virtualizing {ARM}{TrustZone}". In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, pp. 541–556.

[JMV01] D. Johnson, A. Menezes, and S. Vanstone. "The elliptic curve digital signature algorithm (ECDSA)". In: *International journal of information security* 1.1 (2001), pp. 36–63.

[JN03] A. Joux and K. Nguyen. "Separating decision Diffie–Hellman from computational Diffie–Hellman in cryptographic groups". In: *Journal of cryptology* 16.4 (2003), pp. 239–247.

[JU12] J. Jarmoc and D. Unit. "SSL/TLS interception proxies and transitive trust". In: *Black Hat Europe* (2012).

[KAS08] V. Kapoor, V. S. Abraham, and R. Singh. "Elliptic curve cryptography". In: *Ubiquity* 2008.May (2008), pp. 1–8.

[Kes03] G. C. Kessler. "An overview of cryptography". In: (2003).

[KKA21] S. Kaur, K. Kumar, and N. Aggarwal. "A review on P4-Programmable data planes: Architecture, research efforts, and future directions". In: *Computer Communications* 170 (2021), pp. 109–129.

[Kob87] N. Koblitz. "Elliptic curve cryptosystems". In: *Mathematics of computation* 48.177 (1987), pp. 203–209.

[Kre+14] D. Kreutz et al. "Software-defined networking: A comprehensive survey". In: *Proceedings of the IEEE* 103.1 (2014), pp. 14–76.

[LKR13] A. Lara, A. Kolasani, and B. Ramamurthy. "Network innovation using openflow: A survey". In: *IEEE communications surveys & tutorials* 16.1 (2013), pp. 493–512.

[Mal+10] A. Malhotra et al. "UDP based chat application". In: *2010 2nd International Conference on Computer Engineering and Technology*. Vol. 6. IEEE. 2010, pp. V6–374.

[McK+08b] N. McKeown et al. "OpenFlow: enabling innovation in campus networks". In: *ACM SIGCOMM computer communication review* 38.2 (2008), pp. 69–74.

[MD14] S. Manimala and A. T. Devapriya. "Detection of vampre attack using EWMA in wireless ad hoc sensor networks". In: *IJISET Int J Innovative Sci Eng Technol* 1.3 (2014), pp. 450–550.

[MEÁ10] V. G. Martınez, L. H. Encinas, and C. S. Ávila. "A survey of the elliptic curve integrated encryption scheme". In: *ratio* 80.1024 (2010), pp. 160–223.

[Mei+13] S. Meier et al. "The TAMARIN prover for the symbolic analysis of security protocols". In: *International conference on computer aided verification*. Springer. 2013, pp. 696–701.

[Mén+22] J. Ménétrey et al. "Attestation Mechanisms for Trusted Execution Environments Demystified". In: *arXiv preprint arXiv:2206.03780* (2022).

[MG+11] P. Mell, T. Grance, et al. "The NIST definition of cloud computing". In: (2011).

[Mil85] V. S. Miller. "Use of elliptic curves in cryptography". In: *Conference on the theory and application of cryptographic techniques*. Springer. 1985, pp. 417–426.

[MK20] M. Moravcik and M. Kontsek. "Overview of Docker container orchestration tools". In: *2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA)*. IEEE. 2020, pp. 475–480.

[MR09] D. Micciancio and O. Regev. “Lattice-based cryptography”. In: *Post-quantum cryptography*. Springer, 2009, pp. 147–191.

[MV04] D. McGrew and J. Viega. “The Galois/counter mode of operation (GCM)”. In: *submission to NIST Modes of Operation Process* 20 (2004), pp. 0278–0070.

[NAG19] M. Noura, M. Atiquzzaman, and M. Gaedke. “Interoperability in internet of things: Taxonomies and open challenges”. In: *Mobile networks and applications* 24.3 (2019), pp. 796–809.

[Nai17] N. Naik. “Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP”. In: *2017 IEEE international systems engineering symposium (ISSE)*. IEEE. 2017, pp. 1–7.

[NBB20] A. Nilsson, P. N. Bideh, and J. Brorsson. “A survey of published attacks on Intel SGX”. In: *arXiv preprint arXiv:2006.13598* (2020).

[Nis92] C. Nist. “The digital signature standard”. In: *Communications of the ACM* 35.7 (1992), pp. 36–40.

[NPH20] P. Nikbakht Bideh, N. Paladi, and M. Hell. “Software-defined networking for emergency traffic management in smart cities”. In: *Vehicular Ad-hoc networks for smart cities*. Springer, 2020, pp. 59–70.

[Ope22] OpenDaylight. *OpenDaylight*. `https://www.opendaylight.org/`. Accessed: 2022-07-11. 2022.

[OS09] R. Overbeck and N. Sendrier. “Code-based cryptography”. In: *Post-quantum cryptography*. Springer, 2009, pp. 95–145.

[PZH13] M. Pearce, S. Zeadally, and R. Hunt. “Virtualization: Issues, security threats, and solutions”. In: *ACM Computing Surveys (CSUR)* 45.2 (2013), pp. 1–39.

[REC15] K. Rose, S. Eldridge, and L. Chapin. “The internet of things: An overview”. In: *The internet society (ISOC)* 80 (2015), pp. 1–50.

[Res+99] E. Rescorla et al. “Diffie-hellman key agreement method”. In: (1999).

[Res18] E. Rescorla. *The transport layer security (TLS) protocol version 1.3*. Tech. rep. 2018.

[RM+06] E. Rescorla, N. Modadugu, et al. *Datagram transport layer security*. 2006.

[ROL18] L. F. Rahman, T. Ozcelebi, and J. Lukkien. “Understanding IoT systems: a life cycle approach”. In: *Procedia computer science* 130 (2018), pp. 1057–1062.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. “A method for obtaining digital signatures and public-key cryptosystems”. In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[Ryu22] Ryu. *Ryu: Build SDN Agilely*. `https://ryu-sdn.org/`. Accessed: 2022-07-11. 2022.

[SA93] N. I. of Standards and T. (T. Administration. *Secure hash standard*. Vol. 180. 1. US Department of Commerce, Technology Administration, National Institute of …, 1993.

[Sch+22] M. Schneider et al. “SoK: Hardware-supported Trusted Execution Environments”. In: *arXiv preprint arXiv:2205.12742* (2022).

[Sch17] J. Schaad. *Cbor object signing and encryption (cose)*. Tech. rep. 2017.

[Sel+19b] G. Selander et al. “Object security for constrained restful environments (oscore)”. In: *Work in Progress* (2019).

[SG18] C. Sharma and N. K. Gondhi. “Communication protocol stack for constrained IoT systems”. In: *2018 3rd International Conference On Internet of Things: Smart Innovation and Usages (IoT-SIU)*. IEEE. 2018, pp. 1–6.

[SHB14b] Z. Shelby, K. Hartke, and C. Bormann. “The constrained application protocol (coap)(rfc 7252)”. In: *Jun-2014 Available online. http://www. rfc-editor. org/info/rfc7252* (2014).

[Sho94] 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.

[SKA18] V. G. da Silva, M. Kirikova, and G. Alksnis. “Containers for virtualization: An overview”. In: *Applied Computer Systems* 23.1 (2018), pp. 21–27.

[SM17] D. Soni and A. Makwana. “A survey on mqtt: a protocol of internet of things (iot)”. In: *International Conference On Telecommunication, Power Analysis And Computing Techniques (ICTPACT-2017)*. Vol. 20. 2017, pp. 173–177.

[SMF21] M. U. Sardar, S. Musaev, and C. Fetzer. “Demystifying attestation in Intel Trust Domain Extensions via formal verification”. In: *IEEE access* 9 (2021), pp. 83067–83079.

[Smi+20] R. Smith et al. “Battery draining attacks against edge computing nodes in IoT networks”. In: *Cyber-Physical Systems* 6.2 (2020), pp. 96–116.

[SNK12] M.-K. Shin, K.-H. Nam, and H.-J. Kim. “Software-defined networking (SDN): A reference architecture and open APIs”. In: *2012 International Conference on ICT Convergence (ICTC)*. IEEE. 2012, pp. 360–361.

[Sti05] D. R. Stinson. *Cryptography: theory and practice*. Chapman and Hall/CRC, 2005.

[SV18] S. Srilaya and S. Velampalli. “Performance evaluation for des and AES algorithms-an comprehensive overview”. In: *2018 3rd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*. IEEE. 2018, pp. 1264–1270.

[SZ12] H. Shahriar and M. Zulkernine. “Mitigating program security vulnerabilities: Approaches and challenges”. In: *ACM Computing Surveys (CSUR)* 44.3 (2012), pp. 1–46.

[Tea02] C. P. Team. “Capability maturity model® integration (CMMI SM), version 1.1”. In: *CMMI for systems engineering, software engineering, integrated product and process development, and supplier sourcing (CMMI-SE/SW/IPPD/SS, V1. 1)* 2 (2002).

[TGS17] M. Tiloca, C. Gehrmann, and L. Seitz. “On improving resistance to Denial of Service and key provisioning scalability of the DTLS handshake”. In: *International Journal of Information Security* 16.2 (2017), pp. 173–193.

[Tol04] A. Tolk. “Composable mission spaces and M&S repositories–applicability of open standards”. In: *Spring simulation interoperability workshop, Arlington (VA)*. Citeseer. 2004.

[TR516] O. TR-521. *SDN Architecture Issue 1.1*. Tech. rep. 2016.

[TRA15] A. Tosatto, P. Ruiu, and A. Attanasio. “Container-based orchestration in cloud: state of the art and challenges”. In: *2015 Ninth international conference on complex, intelligent, and software intensive systems*. IEEE. 2015, pp. 70–75.

[TSP17] C.-C. Tu, J. Stringer, and J. Pettit. “Building an extensible open vswitch datapath”. In: *ACM SIGOPS Operating Systems Review* 51.1 (2017), pp. 72–77.

[TTW17] P.-Y. Ting, J.-L. Tsai, and T.-S. Wu. “Signcryption method suitable for low-power IoT devices in a wireless sensor network”. In: *IEEE Systems Journal* 12.3 (2017), pp. 2385–2394.

[Tu+21] W. Tu et al. “Revisiting the Open VSwitch Dataplane Ten Years Later”. In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. SIGCOMM ’21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 245–257.

[Tur08] J. M. Turner. "The keyed-hash message authentication code (hmac)". In: *Federal Information Processing Standards Publication* 198.1 (2008), pp. 1–13.

[UM05] M. Uschold and C. Menzel. "Semantic Integration & Interoperability Using RDF and OWL, W3C Editor's Draft 3 November 2005. Rapport technique". In: *World Wide Web Consortium (W3C)* (2005).

[Wan+10] L. Wang et al. "Cloud computing: a perspective study". In: *New generation computing* 28.2 (2010), pp. 137–146.

[Wel08] S. Welberg. "Vulnerability management tools for COTS software-A comparison". In: *Hg. v. University of Twente, online verfügbar unter http://doc. utwente. nl/64654/1/Vulnerability_management_tools_for_COTS_software_-_a_comparison_v2* 1 (2008).

[Wer+19] J. Werner et al. "The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves". In: *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 2019, pp. 73–85.

[WHF03] D. Whiting, R. Housley, and N. Ferguson. "Counter with cbc-mac (ccm)". In: (2003).

[WMY18] L. Waked, M. Mannan, and A. Youssef. "To intercept or not to intercept: Analyzing tls interception in network appliances". In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. 2018, pp. 399–412.

[YS+16] M. B. Yassein, M. Q. Shatnawi, et al. "Application layer protocols for the Internet of Things: A survey". In: *2016 International Conference on Engineering & MIS (ICEMIS)*. IEEE. 2016, pp. 1–4.

Included Publications

HAVOSS: A Maturity Model for Handling Vulnerabilities in Third Party OSS Components

Abstract

Security has been recognized as a leading barrier for IoT adoption. The growing number of connected devices and reported software vulnerabilities increases the importance firmware updates. Maturity models for software security do include parts of this, but are lacking in several aspects. This paper presents and evaluates a maturity model (HAVOSS) for handling vulnerabilities in third party OSS and COTS components. The maturity model was designed by first reviewing industry interviews, current best practice guidelines and other maturity models. After that, the practices were refined through industry interviews, resulting in six capability areas covering in total 21 practices. These were then evaluated based on their importance according to industry experts. It is shown that the practices are seen as highly important, indicating that the model can be seen as a valuable tool when assessing strengths and weaknesses in an organization's ability to handle firmware updates.

Pegah Nikbakht Bideh, Martin Höst, and Martin Hell. "HAVOSS: A Maturity Model for Handling Vulnerabilities in Third Party OSS Components". In *the 19th International Conference on Product-Focused Software Process Improvement, PROFES 2018, Wolfsburg, Germany*, pp. 81-97. Springer, Cham.

1 Introduction

Software maintenance focuses on issues such as adapting systems to changed functionality requirements, changed environments and corrections of faults. Identified security vulnerabilities can be seen as one type of fault that has received more attention in the last decade, following a number of attacks impacting large organizations, customers, and the society. Attacks have also become more sophisticated and the increasing number of software intensive systems, in IT systems and in the shift towards cloud computing and IoT devices, have resulted in more attack targets. The number of security vulnerabilities reported and recorded in the NVD CVE (National Vulnerability Database, Common Vulnerabilities and Exposures) database has for many years been relatively stable, ranging between approximately 4,200 to 7,900 between 2005-2016 [NIS]. In 2017 the number increased to about 14,700. New companies producing connected devices, as well as older mature companies adjusting their products to the current trend of connectivity, increases the competition on the market. Meeting this competition requires high functionality and fast time-to-market. Using open source software (OSS) instead of developing all code in-house decreases development time, and software maintenance for OSS can focus more on updating the software when new versions are released. However, urgent updates as a result of security vulnerabilities can be very costly for the organization. Thus, it is important to handle the process of identifying and evaluating new vulnerabilities, and subsequently update the software, in an accurate and efficient way.

A maturity model can be seen as a tool that helps organizations improve the way they work, typically by introducing and implementing changes in the organization. This is often a slow process, requiring efforts and resources throughout the organization. The change, to be effective, must have support both from management and the work force, and internal communication processes must be well implemented in order to support the change management required for implementing improvements. The maturity model will help the organization identify the issues in need for improving and prioritizing the efforts. It will also help the organization in ensuring that no important aspects are neglected. However, it will typically not in detail describe *how* the changes should be implemented since this can vary widely between organizations and depend on size, type of organization, business domain, regulations etc.

The goal of this paper is to define, evaluate, and present a maturity model (HAVOSS – HAndling Vulnerabilities in third party OSS) focusing on managing vulnerabilities in third party libraries and code, and the subsequent software update activities that are required to limit a product's exposure to attacks. We target all practices related to this aspect of software maintenance for embedded systems. The model builds upon existing models for software maintenance and security, interviews with industry, and recently published guidelines and recommendations for security in IoT devices. An initial version has been iterated using feedback from

industry representatives, and has then been evaluated by industry. The evaluation shows that the proposed practices are highly relevant.

The paper is organized as follows. In Section 2 we first present related work and maturity models focusing of secure software. In Section 5, we describe the methodology used when defining and evaluating the model, and in Section 5 the different maturity levels are defined. Then we present the results of our evaluation in Section 6 and the paper is concluded in Section 7.

2 Related Work

The Capability Maturity Model for Software, CMM, and CMMI (e.g. [SEI08]) have been very influential in how to support process improvement in software engineering. The models guide an organization through five maturity levels where process standardization (level 3) is seen as more mature than project level processes (level 2), and experience based improvement (level 4 and level 5) is a natural continuation after that type of standardization. The idea of standardizing approaches in the organization, and after that to improve through experiences, has influenced the model presented in this paper. There is also the Software maintenance maturity model (SMmm) [Apr+05] addressing unique activities of software maintenance, and there are maturity models for process improvement implementation [NWZ05].

There are several well-known maturity models focusing on software security and the software development life cycle.

The Building Security in Maturity Model (BSIMM) [MMW15] is based on actual practices in a large number of companies. It thus represents the current state of software security. It can be used to assess the Secure Software Development Lifecycle (SSDL). BSIMM covers 12 practices divided into the four main domains *Governance*, *Intelligence*, *SSDL Touchpoints*, and *Deployment*.

OWASP Software Assurance Maturity Model (SAMM) [Cha17] is an open framework developed by OWASP, with the aim to help organizations evaluating their existing software security practices throughout the whole organization. SAMM is a flexible model that is designed to be utilized by both small, medium, and large companies. SAMM is built on business functions of the software development life cycle, and each business function is tied to three security practices. The business functions are *Governance*, *Construction*, *Verification*, and *Operations*.

The Systems Security Engineering – Capability Maturity Model (SSE-CMM) [ISO08] is intended to be used as a tool to evaluate and assess security engineering practices. It allows organizations to establish confidence in their own capability, but it also helps customers to evaluate a provider's security engineering capabilities. The model is based on the idea that organizations need a repeatable, efficient and assured mechanism to improve their security engineering practices. SSE-CMM has been developed to address these needs by reducing the cost of delivering secure

systems. The model contains a number of base practices which are organized into in total eleven process areas.

The Microsoft Security Development Lifecycle (SDL) [Mic10] is another security assurance process focusing on secure software development. The purpose of SDL is to reduce the number and severity of vulnerabilities in software and it aims to guarantee security and privacy during all phases of the development process. Education, continuous process improvement, and accountability are three main concepts of SDL which emphasizes ongoing activities within the whole software development lifecycle. SDL is built upon five capability areas which correspond to different phases of the software development lifecycle, and each area consists of a collection of security activities. SDL defines four levels of maturity for these areas, namely Basic, Standardized, Advanced, and Dynamic. The basic level means little or no processes related to the activity, while dynamic level corresponds to complete compliance across an entire organization.

The Cybersecurity Capability Maturity Model (C2M2) [Chr14] is designed to help organizations of any type and any size to evaluate and improve their cybersecurity programs. The model can be used to strengthen cybersecurity capabilities and also to prioritize actions to improve organization's cybersecurity processes. The model is organized into 10 domains and each domain has a set of cybersecurity practices. Practices in each domain will help organizations to achieve more mature capability in the domain.

The most important features for vulnerability handling such as vulnerability identification, vulnerability assessment, vulnerability tracking and disclosure policy are included in some of mentioned maturity models. Vulnerability identification through software development process exists in BSIMM [MMW15], SAMM [Cha17], SDL [Mic10] and SSE-CMM [ISO08] and only in SMmm [Apr+05] it exits in maintenance phase. Assessing vulnerabilities only includes in SSE-CMM [ISO08]. Vulnerability tracking by incident response team exists in almost all of them. None of them has any communication or disclosure policy except C2M2 [Chr14]. We tried to gather all of these vulnerability handling features in our maturity model and make a complete maturity model for vulnerability handling. Being highly focused on handling vulnerabilities in third party code, our proposed maturity model should not be seen as a replacement for the other models. HAVOSS is intended to be used as a complement to other, more general, maturity models.

3 Methodology

The model has been designed iteratively based on feedback from presenting it to practitioners in the field. A first problem-understanding was achieved through an interview study with practitioners [Hös+18] where it was clear that there is a need to support these processes in industry. In that study, 8 companies in the IoT domain were interviewed about how they handle vulnerabilities in OSS and

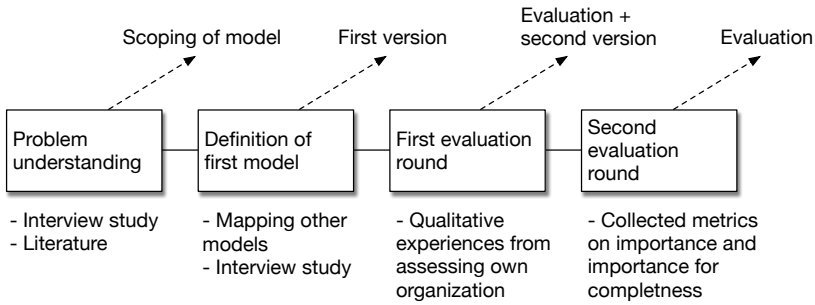


Figure 1: Research steps

COTS code in their developed and maintained products, and what challenges they see in that. It was clear that there is a need to support these activities, meaning that the scope of the model was decided to include all activities that are relevant to identifying and solving vulnerabilities in third party (OSS and COTS) code. A literature study with a comparison to available models also showed the need for this type of model.

3.1 Research Steps

The maturity model was defined through a series of research steps as described in Fig. 1. Based on the identified need, a first version was designed. One important source was the previously conducted interview study with industrial practitioners on how they handle vulnerabilities [Hös+18]. In that study it was clear that many organizations do not have defined processes, neither for identifying, analyzing, or taking care of vulnerabilities in third party code in the products they develop and support. Another input source was already available models, as presented in Section 2. Many of the models include aspects that are related to the capability areas in our model. However, the available models are more general and not as complete in managing third party software vulnerabilities as this model. For example, BSIMM [MMW15] includes "Software Environment" which is related to product knowledge in our model, and it includes "Configuration Management & Vulnerability Management" which is related to evaluating and remedy of vulnerabilities in our model. It is similar for the other models. They include relevant areas, but they are not as focused on vulnerability management for included third-party software where sources of vulnerabilities must be identified and monitored. Based on these input sources, a first version of the model was defined.

The model was decided to consist of *capability areas*, each consisting of a set of *practices* that can be used to identify improvement proposals in assessments. Each practice is represented as a *question* in order make it easier to interpret in an assessment. The final resulting capability areas and questions are presented in Section 4.

When the first version was available it was iteratively improved through evaluations with practitioners, in two main evaluation rounds. Helgesson et al. [HHW11] identify three ways of evaluating maturity models when they are designed, either off-line by the authors alone, off-line by including practitioners, or on-line, i.e., in actual process improvement. Both evaluation rounds in this study can be classified as off-line by including practitioners, since all evaluations are carried out based on practitioners' opinions and experiences of trying to assess their organization. However, at this stage we have not actually conducted any improvement activities guided by the model where a before/after scenario could be analyzed.

In the first evaluation round, refinement of the model was conducted through feedback from practitioners. This was done in several sub-steps, where we in each sub-step sent the model to a contact person in industry who individually assessed their own processes with the model. When they had done that we had a meeting with the organization where we discussed general feedback on the model and we discussed a number of feedback questions, e.g. about if there were any misconceptions from researchers, if the questions were hard to answer, if there were any questions missing, and if the respondent had any thoughts about the answer alternatives. All meetings were held in a way resembling a semi-structured interview where audio was recorded, so the information could be accessed in the analysis. This type of feedback was obtained from two companies, which resulted in a number of adaptions of the model.

In the second evaluation round, feedback was received with other feedback questions than in round 1, now focusing more specifically on every practice of the model. As in the first evaluation round, the model was sent to practitioners, but in this step they were asked to consider not just the answer of each question, but they were also asked to assess the practice with respect to the following two dimensions:

- *Importance of practice*: For each question the participant was asked to judge how important the practice described by the question is in management of vulnerabilities. Possible answer alternatives were 1 – 5.

- *Importance for completeness*: For each question the participant was asked to judge how important it is to include the practice for the completeness of the questionnaire. Possible answer alternatives were 1 – 5. The given motivation was that some practices can be overlooked if they are not included in a model like this. A high score represents that the practitioner thought that it is easily overlooked if it is not included in the model. In the same way a low score means that the practice would probably be solved also without a model like this, i.e. the practice can be considered "obvious".

For each question in the model the participants were allowed to give free text comments in a separate field in the form they received.

The conducted research was influenced by design science. Compared to the framework according to Hevner et al. [Hev+04] it identified the needs and the

Table 1: Participating practitioners

Company	A	B	C	D	E	F	G
Evaluation round 1	✓	✓					
Participants in evaluation round 2	12	2	4	1	1	1	2

problems in the environment e.g. through the interview study, and the evaluations were conducted with people from the same environment. The developed model was, as described above, based on available models and it represents a contribution to the available knowledge base.

3.2 Participating Companies and Practitioners

The participants in evaluation round 1 and evaluation round 2 are summarized in Table 1. The second row shows if the company participated in evaluation round 1 (✓ = yes) and the third row shows how many practitioners from each company who individually answered the questions on importance and importance for completeness in evaluation round 2. The companies are working with software engineering and they represent different size, age, and maturity. Companies A, D, and F are large companies, while the other are smaller. Company E is an example of a startup while the other companies are more traditional companies. Company G offers consultancy services to other companies, while the other companies work with traditional in-house development. All companies but company C are working in the area of embedded software for IoT systems. All involved practitioners were in some way responsible for security and/or working with security-related questions in the organization. In company A most communication was held with the main security responsible. Other persons were working within development roles.

3.3 Validity

The goal has been to obtain good validity of the research by considering typical threats to validity (e.g., [RH09]). Construct validity denotes to what extent the operational constructs are interpreted in the same way by the researchers and the involved practitioners. Care was taken to use as general terms as possible, not to focus on wrong specific meanings of terms in the organizations. This risk can never be ruled out completely, but it can be noticed that some terms in the model were changed in the first evaluation round, in order to not give too specific (and not completely right) meaning to the company practitioners.

Reliability denotes to what degree the analysis is dependent on the specific researchers. This is always a threat, but care has been taken to do the analysis in the whole group of researchers. The analysis has also considered feedback from

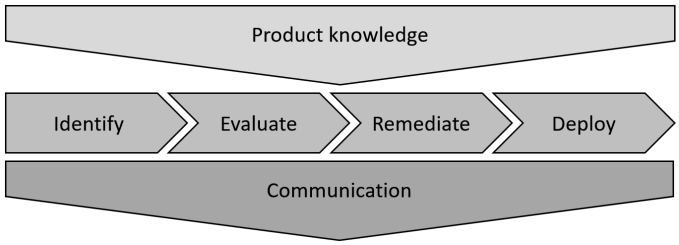


Figure 2: The capability areas included in the proposed maturity model.

members of the industrial participants. For example, both company A and B were involved in both the first end second evaluation round.

Internal validity denotes the risk that other factors than the ones known by the researchers affect the result. This is not a typical controlled study where factors are controlled, but still there may be some factors that affect the results such as ongoing and general improvement attempts with respect to security. Care has been taken to understand the situations of the participating organizations, and many of them have been involved in previous research studies with the researchers. Basically, we see the situation of the participating companies as typical examples of industrial organizations, and no major internal threats.

External validity denotes the possibility to generalize the results. All organizations are Swedish or has a Swedish origin and all participants are employed in Sweden, but they operate on an international market and most of them have mainly international customers. We do not classify them as particularly typical, but more as general examples of organizations in general, at least in the area of embedded systems and IoT systems, when it comes to their approach to managing vulnerabilities. However, in this type of study care must be taken when generalizing to other organizations.

4 Capability Areas and Practices

In this section our proposed vulnerability management maturity model is presented in detail. The six capability areas consist of in total 21 practices. In the assessment sheet, the practices are formulated as questions, e.g. A1, "Tracking maintained and used products" is formulated as "How do you keep track of which type of products are maintained and/or used?"[1]. The capability areas are *product knowledge*, *identification and monitoring of sources*, *evaluating vulnerabilities*, *remedy of vulnerabilities*, *delivering updates* and *communication*. The areas and the relation between the areas is depicted in Fig. 2. Product knowledge is a prerequisite for

[1]The assessment sheet, together with evaluation data are available at `https://zenodo.org/record/1340623#.W2wP7RixWkB`

the other areas and practices. Without this, it is not possible to efficiently, or even at all, handle vulnerabilities. Identifying, evaluating, and remediating vulnerabilities, as well as deploying updates, can be seen as areas of practices that are carried out in sequence. Finally, communication of vulnerabilities and related information can, and often should, be done in parallel with the practices and activities in the other areas. In the following subsections, each capability area and the practices are given in more detail.

4.1 Product Knowledge

Product knowledge assesses companies' knowledge of their products' components. A higher maturity level in this area indicates higher knowledge about the components. This capability area is divided into five practices:

A1. Tracking maintained and used products. Organizations should track maintained products by themselves and also products used by customers regularly, in order to be able to identify their active products.

A2. Tracking included third party OSS and COTS components included in products. Developing companies use many OSS components. This reduces the time-to-market and allow a more cost-efficient development and maintenance organization. Development is largely reduced to selecting the appropriate component to use, while maintenance is reduced to updating it when needed.

A3. Tracking used OSS or COTS versions in the included components. In addition to tracking used OSS and COTS components, it is also of importance to track the versions used in released products and firmware. Version tracking allows an efficient identification of potential vulnerabilities.

A4. Tracking possible threats that products are facing. Threats are possible dangers that might exploit a vulnerability in software products. To avoid critical dangers, and to facilitate correctness in the evaluation of vulnerabilities, it is necessary to track possible threats in software products.

A5. Specifying product usage, operating environment, and end-of-life. By specifying intended usage and operating environment, customers can better understand the intended use of a product, and it also provides important parameters when evaluating the threats and identified vulnerabilities. Specifying an end-of-life informs customers the duration for which they can expect feature and security updates for products. Note that end-of-life for feature updates and security updates can differ.

4.2 Identification and Monitoring of Sources

New vulnerabilities are found on a daily basis and there are several sources for information regarding these. A well defined and efficient process for identifying and monitoring sources of vulnerability information allow both faster and more robust management of vulnerabilities and maintenance of products and devices. The practices in this capability area focus on three aspects.

B1. Determining external sources to use for identifying new vulnerabilities. New vulnerabilities are typically recorded and identified through the CVE numbering scheme [Mit] and further detailed in NVD [NIS]. While this centralized database contains most vulnerabilities, and some other information related to them, it is also worthwhile to monitor new academic results through conference proceedings and journals, as well as monitoring security advisories and the most well-known mailing lists where software security and vulnerabilities are discussed.
B2. Receiving and following up on vulnerabilities reported to the company by external parties. In some cases, new vulnerabilities are disclosed directly to the organization. This can be the case if a third party researcher or professional analyzes the product and reports the results to the manufacturer through a responsible disclosure process.
B3. Monitoring the identified sources of vulnerabilities. Having a well defined process for monitoring vulnerability sources will help minimize the exposure time for products and devices. Often, there are exploits widely available either at the time of disclosure or very shortly after [SSL12].

4.3 Evaluating Vulnerabilities

The goal of this capability area is to help organizations assess their maturity in evaluating the severeness and relevance of identified vulnerabilities. This has direct impact on the next area (remedy of vulnerabilities). Accurate and efficient evaluation, as well as well-founded and correct decisions regarding vulnerabilities, are prerequisites for timely and cost-efficient remediation. The practices in this area thus focus on the following two aspects.
C1. Evaluating severity and relevance of vulnerabilities. After identifying a potential vulnerability, it must be evaluated with respect to product configuration, operating environment, and threat assessment. Unused component functionality, network configuration or unrealistic exploit requirements might render the vulnerability unexploitable. Methods for ranking vulnerability severity might aid in the evaluation. A well-known metric is the Common Vulnerability Scoring System (CVSS) [FIR; MSR07], which gives a rating on the scale 0–10.
C2. Making decisions for handling and reacting to identified vulnerabilities. Firmware and software is often updated on a regular basis in order to include new functionality and patch bugs. Severe security vulnerabilities might need immediate attention and result in updates that are not within the planned cycle. Such updates are very costly and often engage large parts of the organization. It is thus very important to only perform out-of-cycle updates if necessary.

4.4 Remedy of Vulnerabilities

Based on the severity, vulnerabilities can be divided into three basic categories, namely those that need urgent changes, those that can be patched in the next

planned release, and those that need no changes or updates. This capability area assesses the maturity level of organizations for handling these categories.

D1. Handling vulnerabilities that need urgent changes. Urgent changes require immediate action and will impact several processes within the organization. The organization should have an action plan for handling this event in order not to cause unnecessary and unforeseen problems.

D2. Handling vulnerabilities that are updated in a planned release. Here, the maintenance organization must make sure that the affected component is patched in the next release.

D3. Handling vulnerabilities that need no changes. When vulnerabilities have been evaluated, and the results show that attacks are impossible or very unlikely, the organization must make sure that this is well documented. If the component is not updated to a patched version, the vulnerability will always be present, so the organization must make sure that it is not unnecessarily evaluated over and over. Moreover, new information might affect the status of a vulnerability. In that case, it must be re-evaluated since updated information (e.g., new exploits), might affect the decision.

4.5 Delivering Updates

After updating the used components with the latest version, or applying patches to the software, the new firmware or updated software must be deployed to the actual devices. This does not only require a communication channel to the devices, but the channel must also be secure, including verifying the authenticity of new software. However, verifying authenticity is not enough, it is also important to make sure that updates are actually installed on devices [CCS13]. This capability area is divided into two activities.

E1. The process of delivering and applying upgrades to deployed products. The update process can be done fully automatically if the devices support that. In some cases, users will be notified of new updates but needs to go through manual steps to apply them. In other cases, new firmware or software is posted on a website, and it is up to the user to identify and apply these patches. Exactly which process is used can be situation dependent. Although a fully automatic approach is typically preferred, requirements on system or device availability, and also privacy concerns, can render such an approach infeasible in some cases. It can be noted that a recent survey [Pow18] based on 2205 users, reported that only 14% have ever updated their router's firmware.

E2. The process of protecting the integrity of patches while they are delivered and deployed. Integrity protection, typically through digital signatures or MACs, is needed to protect from malicious updates being installed on devices. This in turn will require a PKI or pre-shared keys.

4.6 Communication

Communicating vulnerability and security information, internally and externally, and have structured ways of doing this, allow a more robust and transparent process. It will make the security awareness more visible and contribute to more secure products. This capability is divided into six practices.

F1. Communicating internally when vulnerabilities are identified and resolved. Informing everyone within the company that is somehow affected by the vulnerability, its evaluation, remediation and deployment, allow a well-managed and structured process for updating the software.

F2. Communicating externally when vulnerabilities are identified and resolved. External communication here means e.g., producing advisories that inform the public that the vulnerability has been identified and solved. It also includes forwarding new information to other manufacturers or providing OSS patches upstream.

F3. Communicating with media when vulnerabilities are handled. Widespread and critical vulnerabilities will often come to the attention of media. Well defined processes for communicating with media can improve how the security work within the company is perceived by the public.

F4. Communicating with important customers about critical vulnerabilities. Very large and important customers might be particularly affected by some vulnerabilities, requiring a heads-up when new vulnerabilities are found. Moreover, attacks that affect important customers can have significant impact on the manufacturer's business. At the same time, such communication is resource consuming, for both parts, so it should only be practiced if necessary.

F5. Informing customers about the patching status of products. In order for customers to verify the security of their products, it should be easy to see which software, versions, and patch levels products have. This is part of what is sometimes referred to as a bill of materials. Processes for delivering such information, perhaps together with specific information related to patched vulnerabilities can ease the burden for the support team.

F6. Transferring other security related information while delivering patches. Attaching information on patched vulnerabilities and also providing information on how the patch should be applied, or which additional configuration settings should be applied, can help the customer understand why the patch is applied.

5 Maturity Levels

The intention of the maturity levels is that they should represent an increasing maturity for the assessed organization when it comes to their processes for working with third-party vulnerability updates. This type of maturity can, of course, be defined in different ways, but as described in Section 2, we have chosen a way of viewing maturity that is inspired by the approach in CMMI for software devel-

Table 2: Maturity levels used in the assessment

Level	Description
0	We don't do this.
1	We do this in an ad-hoc way based on individual's own initiatives.
2	We know how we do this, but we do it in different ways in different teams/products.
3	We have defined processes for this that are common to all teams/products.
4	We collect experience and/or metrics from our approach and base improvements on that.

opment. This means that an increasing maturity implies an increased definition and standardization of approaches in the organization. We argue that this standardization is necessary in order to be able to learn from experiences and also to be effective in managing vulnerabilities. If different parts of an organization have individual responsibility to define and manage their processes for this it will not be effective. This means that we can formulate the basic contents of the levels as follows.

The first level is *level 0*, which means that no effort is spend at all on the activity. It may be that an organization does not work with vulnerabilities at all. Then they are assessed at this level. The next level, resembling level 1 in CMMI, *level 1* means that the process is carried out in some way but it is often unclear how it is done, and the responsibility is often left to developers who happen to find the need and have the right competence and resources for it. At the next level, *level 2*, there are defined approaches and routines, although there is not a standardized approach in the organization. The next level, *level 3*, represents a state where there is a standardized process in place for the practice. That is, the same, defined, procedures are used in all teams and projects. At the most advanced level, *level 4*, experiences are collected from using the standardized procedures, and these experiences are used when constantly improving the processes.

In the model presented to the participants, the maturity levels were presented as described in Table 2. When performing an assessment of the maturity, the intention is that every question is assessed. That is, there is one assessment result (level 0 – level 4) for each question. The results can then be presented either as one result for each question or a summary for each area of questions.

When improvements are identified based on an assessment it is possible to identify improvements based on the questions with low scores. When this is done there are some dependencies that can be identified. It is, as described in Section 4, possible to see that capability area A about product knowledge is a pre-condition for the other capability areas, see Fig. 2. It is therefore recommended to start with capability area A in an improvement programme.

6 Results of Evaluations

In this section the results of carrying out the evaluations are presented.

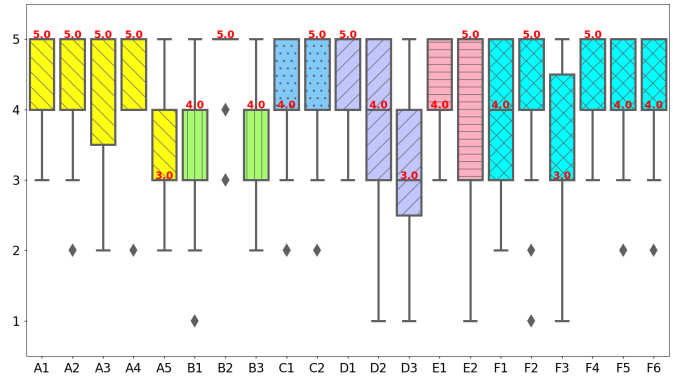


Figure 3: Importance of activity

6.1 First Evaluation Round

In the first evaluation round a number of adaptions were made. In the discussions it was clear that the practitioners thought that there were no major misconceptions, and that the model included the major important aspects according to them. However, it was clear that some terminology that was used could be changed to terms that are more general in order to lower the risk of confusion about company specific terms. There were also some questions, especially in capability area A that were refined in order to be more understandable. Concerning the completeness, new questions about how to communicate with external sources, such as customers, were added. Also, based on the question about answer alternatives, i.e. the maturity levels, they were presented in a clearer way and the two highest levels in that version of the model were combined into the current most advanced level. In the original version there was one level for collecting experience and another level for using the experiences for improvement. These changes resulted in the model that is presented in this paper (Section 4 and Section 5).

6.2 Second Evaluation Round

In the second evaluation round the focus was on understanding the important of the questions and to what extent the questions would be handled without any model. The results with respect to *importance of activity* and *importance for completeness* of each question are shown in Fig. 3 and Fig. 4. Median values have been explicitly given to avoid ambiguity in the plots.

It can be observed that almost all questions are seen as important by the practitioners. The freetext answers reveal some more detailed perceptions. One comment on D3 (Handling vulnerabilities that need no changes) was that this might be easily overlooked. This captures the importance of the question but also indicates why it has received slightly lower score overall. It is not seen as important as

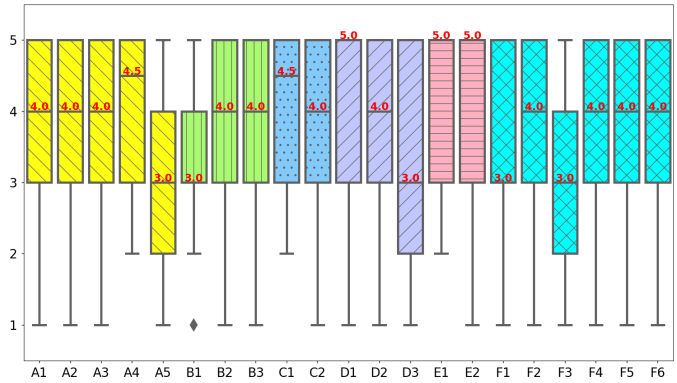
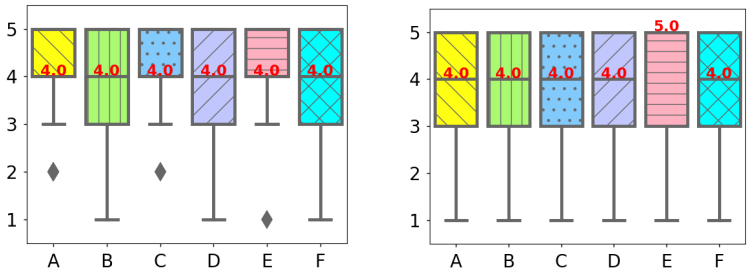


Figure 4: Importance for completeness



a) Importance b) Importance for completeness

Figure 5: Grouped results

vulnerabilities that do require changes. Question F3 (Communicating with media), which also had a relatively low score, was not present in the initial version. It was added after interviewing company A, who viewed this as an important aspect that was not covered by the other questions. One comment on this question (from another company) was that this is mostly relevant for larger companies.

Some freetext answers also suggested adding more questions. One suggestion was to add security assessment, in which assets are identified. The importance of such a question will depend on to which extent the company knows which assets are actually protected. Another suggestion was to also consider how third party components are selected. Components, and in particular their maintainers must be trusted not to e.g., add malicious code into the software. To see if there are differences between the capability areas, we aggregate the answers to these areas, see Fig. 5. Again, it can be seen that the values are high, and there are only minor differences between the areas. A Kruskal-Wallis test (non-parametric alternative to one factor, n-levels, ANOVA) shows that no significant difference could be found between the areas ($p = 0.23$ for importance), which can be expected from the

graphs.

Approximately half of the evaluation answers were from company A. We compared the results from company A to the results from the other companies by looking at box-plots and it seems they are not different. This is a motivation why we analyze the results from every respondent without considering the company. These differences were also analyzed for each question with Mann-Whitney tests, but as expected no significant differences were found. Company A was shown a summary of their responses and asked if they think their result would have differed 2 years back or 2 years from now.

> Not significantly different. If we look back a bit further, say 5 years, most activities have definitely increased in importance. Some activities will probably increase a bit further in the future, especially the F-section where laws and regulations might play a part, but overall the activities are already perceived as important. Reduced importance is unlikely in the foreseeable future.

They were further asked if the similarity between Company A and other companies were expected.

> It's expected. It indicates the increased attention to security issues is not restricted to specific businesses and this is what we have perceived as well.

That is, it can be seen that company A are working with improvements that are in line with the model.

7 Conclusions

The presented maturity model aims to help organization assess their maturity in handling software vulnerabilities in third party OSS and COTS components. The importance of such a model is due to the increasing number of vulnerabilities that are being reported, and the growing number of connected devices that are bound to change the society in the near future. The model is based on six capability areas and 21 practices. Related maturity models, i.e., those that focus on software security are very broad and cover many aspects related both to software development, maintenance, and organizational aspects, but they are not detailed enough to cover all aspects of handling vulnerabilities in third party code. Thus, this model can be seen as an important complement to other well-known models. This is also supported by our evaluation, which shows that the defined practices are highly relevant.

Acknowledgements

This work was supported partly by the Wallenberg AI, Autonomous Systems and Software Program (WASP) and partly by Swedish Governmental Agency for Innovation Systems (Vinnova), grant 2016-00603.

References

[Apr+05] A. April et al. "Software Maintenance Maturity Model (SMmm): the software maintenance process model". In: *Journal of Software: Evolution and Process* 17.3 (2005), pp. 197–223.

[CCS13] A. Cui, M. Costello, and S. J. Stolfo. "When Firmware Modifications Attack: A Case Study of Embedded Exploitation." In: *Network & Distributed System Security Symposium*. 2013.

[Cha17] P. Chandra. *Software Assurance Maturity Model - A guide to building security into software development*. Tech. rep. OWASP, 2017.

[Chr14] J. D. Christopher. *CYBERSECURITY CAPABILITY MATURITY MODEL (C2M2)*. Tech. rep. Rhodes University, 2014.

[FIR] FIRST. *Common Vulnerability Scoring System v3.0: Specification Document*. https://www.first.org/cvss/specification-document, Last accessed 2018-06-03.

[Hev+04] A. R. Hevner et al. "Design Science in Information Systems Research". In: *MIS Quarterly* 28.1 (2004), pp. 75–105.

[HHW11] Y. L. Helgesson, M. Höst, and K. Weyns. "A review of methods for evaluation of maturity models for process improvement". In: *Journal of Software Maintenance and Evolution: research and Practice* 24 (2011), pp. 436–453.

[Hös+18] M. Höst et al. "Industrial Practices in Security Vulnerability Management for IoT Systems – An Interview Study". In: *Proceedings of Software Engineering Research and Practice (SERP)*. 2018.

[ISO08] ISO/IEC. *Information technology — Security techniques — Systems Security Engineering — Capability Maturity Model*. Tech. rep. International Organization of Standardization, 2008.

[Mic10] Microsoft. *Simplified Implementation of the Microsoft SDL*. Tech. rep. Microsoft Coporation, 2010.

[Mit] Mitre. *Common Vulnerabilities and Exposures*. https://cve.mitre.org/. (visited on: 2018-05-15).

[MMW15] G. McGraw, S. Migues, and J. West. "Software security and the building security in maturity model (BSIMM)". In: *Journal of Computing Sciences in Colleges* 30.3 (2015), pp. 7–8.

[MSR07] P. Mell, K. Scarfone, and S. Romanosky. "A complete guide to the common vulnerability scoring system version 2.0". In: *Published by FIRST-Forum of Incident Response and Security Teams*. Vol. 1. 2007, p. 23.

[NIS] NIST. *National Vulnerability Database*. https://nvd.nist.gov/. (visited on: 2018-05-15).

[NWZ05] M. Niazi, D. Wilson, and D. Zowghi. "A maturity model for the implementation of software process improvement". In: *Journal of systems and software* 74.2 (2005), pp. 155–172.

[Pow18] M. Powell. *Wi-Fi router security knowledge gap putting devices and private data at risk in UK homes*. Tech. rep. Available at https://www.broadbandgenie.co.uk/blog/20180409-wifi-router-security-survey.
2018.

[RH09] P. Runeson and M. Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical Software Engineering* 14 (2009), pp. 131–164.

[SEI08] SEI. *Capability Maturity Model Integration, Version 1.2*. Vol. CMU/SEI-2006-TR-008(2008). Carnegie Mellon Software Engineering Institute, 2008.

[SSL12] M. Shahzad, M. Z. Shafiq, and A. X. Liu. "A large scale exploratory analysis of software vulnerability life cycles". In: *Proceedings of International Conference on Software Engineering (ICSE)*. 2012, pp. 771–781.

Energy Consumption for Securing Lightweight IoT Protocols

Abstract

In this paper we address the energy consumption of the Constraint Application Protocol (CoAP) and the Message Queue Telemetry Transport (MQTT) protocol and compare their overhead. We also pay attention to the use case of security in IoT and analyze the energy consumption when using TLS/DTLS for the two protocols. In our experiments we use ESP32 with libcoap, MQTT, and mbed TLS libraries and conduct real-world measurements using Otii, a high precision voltage and current measurement tool. While the particular numbers are implementation and hardware dependent, we can still make interesting observations. For data transfer, we find that aggregating data to larger packets can significantly reduce the energy consumption. We also find that AES-CCM8 seems slightly more efficient than other modes of operation. In comparison, the DTLS handshake for setting up the secure connection is very expensive, and also very dependent on security level and algorithm choices. For firmware updates, AES-CCM8 is again slightly better than the alternatives, but the differences between CoAP and MQTT are much more significant, favoring MQTT due to the use of the retransmission support in TCP. This is also evident in lossy networks, where MQTT saves up to 91% energy compared to CoAP at 20% loss rate. Finally, we find that energy consumption in CoAP can to some extent be reduced in lossy networks by modifying

Pegah Nikbakht Bideh, Jonathan Sönnerup, and Martin Hell. "Energy Consumption for Securing Lightweight IoT Protocols". In *proceedings of the 10th International Conference on the Internet of Things, IoT 2020, Malmö, Sweden*, pp. 1-8, ACM.

the retransmission timeout.

1 Introduction

Devices connected to the Internet of Things (IoT) are seen as key enablers for e.g., the smart city, home automation, wearables, and asset tracking. Connected devices will supposedly also improve or even revolutionize, among other things, energy management, healthcare and the management of our infrastructure. The devices will be realized as e.g., sensors for detecting and monitoring physical characteristics of the environment, or actuators to control our environment, machines, systems or processes. Their interconnection with other devices, gateways and/or the cloud introduces new security challenges, but it also makes them more exposed to attacks, targeting e.g., unpatched vulnerabilities. The most common application level communication protocols for IoT are the Constraint Application Protocol (CoAP) and the Message Queue Telemetry Transport (MQTT) protocol. CoAP is a lightweight protocol, in many aspects similar to HTTP, while MQTT is a publish/subscribe protocol. Both protocols are widely supported and have gained widespread adoption, but neither include security functionality. Still, support for confidentiality and integrity of messages, as well as message authentication, is often needed, and the natural choice is then to use DTLS for CoAP and TLS for MQTT.

The ubiquitous nature of IoT devices often requires them to run on batteries, making energy efficiency a primary concern. The large number of devices make it costly to replace batteries, and it will also make the total energy consumption considerable, further motivating energy efficient data communication and processing. At the same time, adding security to the communication will add additional overhead. Thus, it is important to not only develop lightweight security protocols, but also to understand to which extent security affects the energy consumption of the devices. Such understanding will allow vendors and users to make informed decisions when choosing and implementing security in the devices and systems.

The main contribution of this paper is a thorough analysis of CoAP and MQTT and the investigation of their energy footprint in different scenarios, and how added security at the transport layer (TLS/DTLS) affects the energy consumption. Important design choices, such as cipher suite, PKI vs. PSK, and client authentication are also analyzed in order to better understand how such choices impact the energy consumption. In our real-world experiments, we use ESP32 to represent a device. For measurements, we use an Otii[1], which is a high precision voltage and current measurement unit Note that the actual numbers given in this paper are implementation and hardware specific and might differ between libraries or IoT devices. Still, the main takeaways will apply to the general case.

[1] https://www.qoitech.com/

2 Related Work

Efficiency and comparisons of IoT protocols have been considered in several previous works. An important thing to note is that the library or implementation used can heavily impact performance [IOU17]. Optimized implementations was pursued for DTLS in [Cap+15], where the authors exploited ECC optimizations in order to minimize ROM occupancy, time and energy. They only considered ECC based operations with two different cipher suites, one with ECDH and ECDSA, and one with ECDHE and PSK. The importance of optimized implementations was also noted in [Suá+18] where it was shown that secp256r1 was more efficient than the secp224r1 curve due to a more optimized implementation. They compared ECDSA and RSA in TLS 1.2 and on ESP32, and showed that ECDSA performs better than RSA.

Performance of security updates for IoT were measured and discussed in [TBK19]. Three different models, CoAP, MQTT, and encapsulating CoAP inside MQTT, was proposed for delivering Over-the-Air updates and software patches. While it was shown that MQTT is faster and more reliable than CoAP to send urgent updates, no energy measurements were made and cryptographic protection was not considered. Since power consumption differs over time, it is not possible to draw accurate conclusions for energy consumption by only measuring time. In this paper, we consider both energy and time for the software update case, while also comparing both CoAP and MQTT using different encryption algorithms.

In [Tha+14], the authors designed a common middleware for MQTT and CoAP and measured performance of these based on end-to-end delay of single packets and bandwidth consumption. Their results indicated CoAP has lower average delay in case of high packet loss, around 25%. No energy measurements were made. In this paper, we measure energy and looks at a sequence of packets instead of average time for individual packets.

Energy can also be reduced by instead making changes to the protocol. Lithe (Lightweight Secure CoAP for the Internet of Things) [Raz+13] is an integration of DTLS and CoAP for IoT, in which the authors proposed a header compression scheme which aims to reduce energy consumption by leveraging 6LoWPAN. In their evaluation, they demonstrated that CoAP overhead by using DTLS compression can be significantly reduced in terms of energy consumption and network response time. While such modifications are valuable, they require protocol modifications. In this paper we do not aim to make modifications to the protocols.

3 CoAP and MQTT

3.1 CoAP

CoAP [GMS15] is a web transfer protocol, for constrained networks and was developed by IETF CoRE (Constrained RESTful Environments) Working Group.

CoAP was designed for UDP communication over 6LoWPAN networks. It uses a Universal Resource Identifier (URI) to identify available resources on constrained devices. Messages are exchanged between endpoints using CoAP requests and responses. CoAP is a one-to-one protocol for transferring information between client and server.

A CoAP message has a 4-byte fixed header, consisting of 2 bits for Version field, 2 bits for message Type, 4 bits of Token Length, 8 bits of Code field and 16 bits of Message ID. The Message ID is utilized for duplicate detection. A token can optionally be used to match requests and responses. In our experiments, we do not include Token or Options, so the Token Length is zero.

The CoAP specification describes four security modes:

- **Nosec**: No security provided.
- **PreSharedKey**: Symmetric pre-shared keys are used in one of the PSK cipher suites in DTLS.
- **RawPublicKey**: DTLS is used with an asymmetric key pair that is pre-installed on the device, without a corresponding X.509 certificate.
- **Certificates**: x.509 certificates are used with DTLS.

For CoAP, we may choose to send either confirmable (CON) messages or non-confirmable (NON). CON messages will be ACKed, or retransmitted in case of packet loss, whereas NON messages will not.

CoAP can also run over TCP [Bor18] instead of UDP, but it is only applicable for cases where there is no other choice for networking infrastructure. It causes larger code and packet sizes, increases RAM requirements, and increases round trips. Therefore, in this paper the original CoAP protocol over UDP is considered.

3.2 MQTT

The MQTT protocol is a publish/subscribe messaging protocol designed for low bandwidth environments, originally for communication over satellite links. It uses a server (broker), together with a set of clients. A client sends messages tagged with a topic, and other clients can subscribe to that topic, in which case the server routes the messages to the subscribing clients. In MQTT, the connections to the server can be specified with a Quality of Service (QoS). QoS can vary from 0 to 2 in which 0 has the least overhead. In our experiments, we used the default QoS, which is 0. MQTT is a many-to-many protocol as it passes the messages between clients through the broker.

MQTT runs over the TCP protocol. There is also a variant called MQTT-SN[2] which can use UDP. It has however not received widespread adoption and is not well supported, so it is not considered in this paper.

[2]http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf

3.3 Security on the Transport Layer

Both CoAP and MQTT can have security added at the application layer. OSCORE (Object Security for Constrained RESTful Environments) [Sel+19a] is an example of an object security protocol for CoAP. Still, the two protocols are commonly used with DTLS and TLS and both specifications explicitly discuss the use of these protocols. TLS and DTLS differ primarily by the fact that DTLS, being used for UDP, must e.g., handle packet loss and out-of-order packets in the handshake phase.

The two main protocols in TLS are the handshake protocol and the record protocol. In the handshake protocol, the peers are authenticated and encryption and message authentication keys are established. A Diffie-Hellman key exchange, or a pre-shared key, is used to agree on a premaster secret, from which the encryption keys are derived. With Diffie-Hellman, the messages are authenticated using a digital signature. It is also possible to combine Diffie-Hellman and PSK, in which case the two values are concatenated to form the premaster secret. For improved performance, Diffie-Hellman can be computed over elliptic curves (denoted ECDHE in the cipher suites). Also the digital signature can be computed over an elliptic curve (ECDSA), instead of using an RSA signature. Once keys have been established in the handshake protocol, data can be encrypted and authenticated in the record protocol.

The record layer in DTLS is very similar to TLS, but an explicit epoch (2 bytes) and sequence number (6 bytes) are added to the record. This results in DTLS 1.2 messages having 29 bytes overhead while TLS 1.2 has only 21 bytes overhead.

Other than normal message overheads, DTLS 1.2 handshake messages have 7 extra bytes header overhead. This overhead in the DTLS 1.2 handshake, in comparison to TLS 1.2, is due to the fact that DTLS handshake messages can be fragmented over several DTLS records.

4 Experimental setup

Here we present the architecture used during the experiments, i.e., the hardware and software components.

4.1 Selected Hardware

To select the development board, based on our requirements including low price, good documentation, support for CoAP/MQTT and TLS/DTLS and widespread use, we decided to use ESP32-WROOM-32D [Esp19]. It is relatively cheap, has support for the software libraries we target, and has good documentation and a large community. It can also be used in many applications ranging from low-power sensor networks to very demanding tasks. Moreover, AES hardware acceleration is supported (and was enabled). It has an Xtensa 32-bit dual-core microprocessor

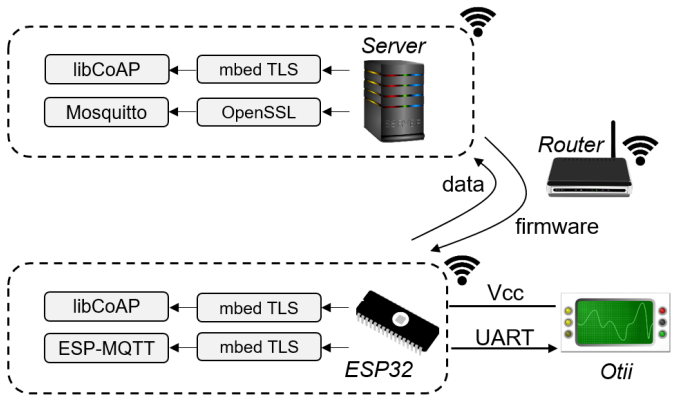


Figure 1: CoAP/MQTT testbed architectural overview

with 240 MHz clock speed and 512 KiB of RAM. We directly supplied ESP32 with 3.3V which was the supported operating voltage. Although, 6LoWPAN is beneficial in low power IoT networks but we have used WiFi for connectivity since 6LoWPAN was not supported on ESP32.

As WiFi router, an Asus RT-ACS1U was used. The CoAP server and the MQTT broker, were run on a 64-bit Linux system with an Intel Core i5-6200U at 2.4 GHz with 8 GB RAM.

To measure energy consumption, an Otii Arc was used. Otii is a portable power supply and data-acquisition module which can be used for very accurate voltage measuring. It is commonly used by developers in device and application designs to optimize energy consumption. Otii Arc has a desktop application available for Windows, Ubuntu and macOS. We used the application on Ubuntu.

4.2 Software

The official development framework for ESP32-IDF v3.3 [Esp20], denoted ESP-IDF, was used for development. The CoAP client was developed with libcoap 4.2.1, and the mbed TLS 2.16.2 library was used to setup the DTLS connection.

The libcoap 4.2.1 library was also used to setup the CoAP server on the Linux machine. For transport layer security, libcoap has support for GnuTLS, OpenSSL and Tinydtls, and can also be configured with mbed TLS. In our testbed libcoap with mbed TLS 2.8 was used.

The MQTT library used on the client side is the one included in the ESP-IDF, running MQTT version 3.1.1. The library utilized mbed TLS for TLS 1.2, supporting both PKI-based and PSK-based cipher suites. On the server side, the Mosquitto broker 1.6.7 is utilized using OpenSSL 1.1.0l. The code for the ESP32

along with used scripts can be found online[3].

To do the measurements with the Otii software, the Otii device is connected to a power supply via a USB port. Then, the Otii device powers the ESP32, allowing it to measure the power used. UART messages are sent from the ESP32 to the Otii in order to annotate the measurements. This allows us to accurately correlate the energy consumption to the different phases of the application under test.

The selected components are connected as shown in Fig. 1.

5 Methodology and Use Case

In this section, we discuss, and give the rationale behind, our chosen measurements. In the following, the cost will refer to the energy cost of first computing, and then sending information over the communication channel.

A typical use case would be a client/server application in which a client or sensor communicating data back to a server for aggregation and further analysis. The sensors are developed using a mix of third party and in-house developed code. They will thus be subject to discovered security vulnerabilities, requiring new firmware with regular intervals. The sensors will typically be powered by a battery. The process of changing the battery is resource consuming, requiring energy efficient computation and communication.

Although TLS 1.3 has faster encryption speed and requires less round trips to complete the handshake in comparison to TLS 1.2, in this paper, we focus on TLS 1.2 and DTLS 1.2. TLS 1.3 was finalized in August 2018 and DTLS 1.3 is only in draft state. The versions are not compatible with each other, so to have a fair comparison between CoAP and MQTT, we do not consider TLS 1.3 for MQTT. Still, we only consider cipher suites that are compatible with TLS 1.3, i.e., we exclude RSA key exchange and we only consider AEAD modes of operation on the record layer.

The main goal is to better understand the following aspects:

- The cost of adding security to CoAP and MQTT and the difference between AES modes of operation and key sizes when encrypting bulk data.
- The handshake cost, using different cipher suites.
- The cost of updating the device firmware, using a secure channel.
- The influence of packet loss for bulk data transfer.

Adding TLS/DTLS at the transport layer will incur overhead for both initial handshake messages and encryption/decryption and authentication of messages in the record layer.

[3] https://github.com/Noxet/squidward

5.1 Adding Encryption to Data

To understand the cost of encrypting data, and how algorithm choices affect the cost, we measure the energy used to encrypt messages of different sizes.

According to the CoAP specification [BZ16], CoAP messages should fit into a single IP datagram to avoid IP fragmentation. Thus, the payload size is bound to 1024 bytes. To transfer larger payloads, CoAP supports a block-wise transfer option. This option enables transferring multiple blocks of information represented as multiple request-response pairs. The block-wise option enables a server to be stateless, since the server handles each block separately and there is no need for any connection setup on server side. In order to cover a wide range of use-cases, we measure the energy for message sizes between 16 and 8192 bytes, i.e., up to 8 blocks.

When measuring the encryption overhead, the TLS/DTLS handshake is not considered, only the encryption in the record layer. The handshake is measured separately, as detailed in the next section.

5.2 TLS/DTLS Handshake

The TLS/DTLS handshake can be based on asymmetric keys or a symmetric Pre-Shared Key (PSK). In the case of asymmetric keys, digital signatures are used to authenticate the key exchange message in the handshake, while in the PSK case it is possible to either directly agree on a PSK to use as pre-master secret, or to use ephemeral Diffie-Hellman key exchange (for perfect forward secrecy), and then adding the PSK to construct the pre-master secret. Since asymmetric operations are computationally expensive, PSK can be favourable in constrained environments. Still, digital signatures can be preferred when it is not feasible to pre-share keys. The RawPublicKey variant is currently not supported in mbed TLS, the library used by our device. We analyse the energy consumption for the handshake in the other modes, PreSharedKey and Certificates. The comparison measures both the difference between the modes and how different algorithms and levels of security compare to each other. Moreover, we compare how client authentication influences the energy and time. All measurements are made for CoAP.

Many cipher suites are available, but we take the NIST guidelines [MC19] into account for the selection of cipher suites. Apart from PSK, we only consider ephemeral keys and Diffie-Hellman key exchange (RSA for key exchange is deprecated). Since mode of operation and cipher algorithm only marginally effects the handshake, we fix these to AES in GCM mode.

We use two approximate security levels of 128 (moderate) and 256 (high) bits. For *moderate* security level we choose elliptic curves of size 224 and 256 bits. For RSA signatures, we then use RSA-2048. CoAP explicitly mandates the use of the curve secp256r1, which uses an elliptic curve with a 256-bit prime (also known as NIST P-256). The curve secp224r1 corresponds to the 112-bit security provided by RSA-2048. For the application data, we use AES-128 together with SHA-256.

Since we use GCM/CCM, the hash algorithm is only used for the PRF in the handshake, not to compute a MAC on the record layer. For *high* security level, we instead use 384- and 512-bit curves, and RSA-4096. We adjust the encryption key to 256 bits and use SHA-384. See e.g., [Bar19] for more information on security levels.

Although the use of PSK is not recommended by NIST, they do list a set of PSK cipher suites that can be used. We compare both plain PSK, i.e., (PSK as premaster secret (`PSK`), and when Diffie-Hellman is used together with the PSK (`DHE_PSK`).

5.3 Firmware Update

A new firmware is often relatively large, at least in comparison to the typical data packets sent by devices. A firmware update process typically consists of:

1. Download the firmware from a server.
2. Store the firmware such that the bootloader can boot into the firmware.
3. Reboot the device, using the new firmware.

The ESP32 has built-in OTA (Over-The-Air) update functionality, combining steps 1 and 2 above. We measure the energy and time used to both download and store a firmware with size 870 KiB. Rebooting is not part of the measurement. We compare different encryption key sizes and modes of operation for AES, for both CoAP and MQTT, in order to analyze the difference for these larger data transmissions.

5.4 Packet Loss

Packet loss can be costly due to retransmission, resulting in more energy consumption. If a CoAP message is marked as CON, it will be retransmitted until the receiver sends an ACK. For MQTT, retransmission is handled by the TCP layer.

We simulate packet loss with loss rate varying from 0 to 20% using CoAP and MQTT, both with and without security. Very high ranges of packet loss may cause a session to be lost due to timeout and ranges of very low packet loss does not have much effect on energy consumption. Also, based on used loss rates defined in [Tha+14], we use packet loss rates up to 20%.

The CoAP messages were marked as CON in our experiments, thus when a packet loss occurs, the packet will be retransmitted until an ACK is received from the server. One might believe that NON messages are more efficient, if packet loss is not a concern, but this is not the case. According to RFC 7252 [SHB14a], CoAP is a request and response protocol, this means that CON and NON messages both

are followed by a response. In the case of a CON message, the ACK is usually piggybacked, resulting in no byte overhead.

In CoAP, the retransmission time is primarily controlled by the `ACK_TIMEOUT` parameter, i.e., the time after which a retransmission is made. According to RFC 7252, the default value is 2 seconds, and it is recommended to not be less than 1 second. Following this, we vary the timeout between 1 and 3 seconds, and analyze its effect on the energy consumption with different packet loss rates.

6 Results and Discussion

In this section we give the results of our measurements. In some measurements, we also measured the duration time since energy and time is not fully correlated. Energy consumption is not necessarily the same for doing the required computations and for sending the actual payload. All our measurement results are given as the average of 10 measurements. The power consumption in the idle state is $108\,mW$, which corresponds to 30 μWh during a second. For the PKI-based handshake, certificates are signed by a CA, so they have a certificate chain of length two. Note that the required energy for WiFi initialization is included in all of the measurements.

6.1 Analysis of Adding Encryption to Data

We analyze the energy consumption for sending data from the client to the server. CoAP and MQTT communication is analyzed, both without security, and with different variants of channel encryption. We start the measurement after the client's finished message in the handshake and stop the measurement when all data has been transmitted.

The total energy consumption for CoAP is given in Fig. 2, while the results for MQTT can be found in Fig. 3. Looking at plain data (NULL), CoAP consumes slightly less energy than MQTT and is more efficient for packet sizes up to 1024 bytes. This is reasonable since UDP is more efficient than TCP over a reliable network. However for larger data chunks (> 1024 bytes), CoAP consumes much more energy, so here MQTT is more efficient. The reason for this can be found on application level. CoAP has a 1024 byte limitation on packets with no support for fragmentation. Thus, CoAP must send several consecutive chunks of 1024 bytes, while MQTT can send larger payloads up to 256MB although fragmentation may occur in lower layers. Further, we can note that the energy cost of sending data is constant for up to around 1 KiB of data for both CoAP and MQTT. The energy needed to initiate and end the actual WiFi communication is then significantly larger than the energy cost of the actual data transmission. For CoAP (UDP), this overhead is around 5.7 μWh (with standard deviation 0.07) and for MQTT (TCP) it is around 6.1 μWh (with standard deviation 0.3).

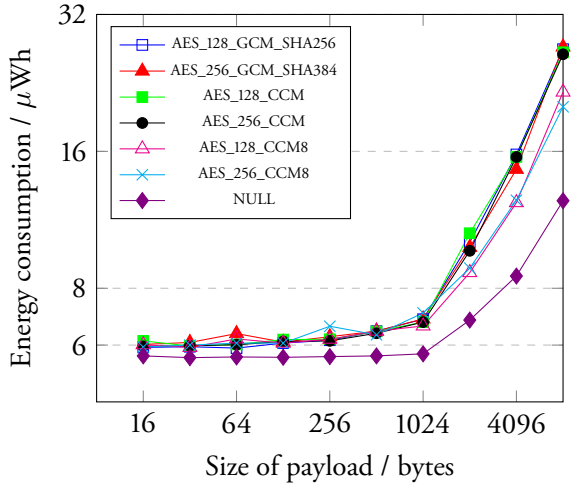


Figure 2: Energy consumption for CoAP/CoAPs protocols with different cipher suites and different payload sizes

From Fig. 2, we can see that adding DTLS to CoAP messages (shown with different cipher suites in Fig. 2) adds a small amount of energy (compared to NULL), in the order of 0.3 μWh, for small messages. For larger messages, the added energy is increasing. Adding TLS to MQTT gives lower overhead compared to DTLS, likely since it has less byte overhead compared to DTLS. For smaller messages the difference is within one standard deviation of the measurements. For larger messages the difference is evident.

Looking at Fig. 2 and 3, with different cipher suites, it is clear that there is not much difference in the choice of keysize (128 vs. 256 bits) and modes of operation (GCM vs. CCM). Calculating a 95% confidence interval shows that there is no statistical difference. One thing that we can notice is that for CoAP messages, the CCM8 variants require (statistically) less overhead compared to the other variants. For messages of size 8192 bytes the energy saving is around 6 μWh.

From Fig. 2 and 3 we can also conclude that aggregating as much data as possible is beneficial for energy consumption. For instance, sending 8 kiB in frames of 16 bytes is much more energy consuming than transferring 8 KiB in frames of 1024 bytes. This is even more important if it can also reduce the number of handshakes, as will be seen in the next section.

Key takeaways:

- Though modes of operation and key sizes changes the cryptographic algorithms, for our used hardware and software libraries, this has very little impact on energy consumption. Still, CCM8 seems to be somewhat cheaper.

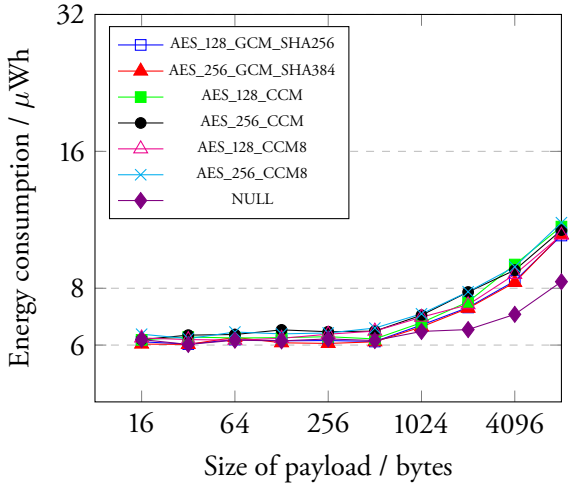


Figure 3: Energy consumption for MQTT/MQTTs protocols with different cipher suites and different payload sizes

- Since CoAP can send packets of size up to 1024 bytes, the overhead of sending several packets makes MQTT less energy consuming for payload sizes larger than 1024.
- In both CoAP and MQTT, sending aggregated data is more beneficial than sending data in smaller packets.

6.2 Analysis of PSK-based and PKI-based DTLS Handshakes

To measure the handshake part of a secure connection in CoAP, we setup a secure connection from the ESP32 to the server, forcing the client to support only one specific cipher suite listed in Table 1. To establish the actual connection, the client sends 16 bytes of data to the server. For each cipher suite, we measure the energy consumption for the handshake, including transferring 16 bytes of data and closing the connection. This is performed both with and without client authentication. The measured values and duration times, are given in Table 1.

For the two PSK-based methods, the addition of Diffie-Hellman (i.e., PFS) will significantly increase the energy needed.

The remaining cipher suites can be compared from different perspectives. Comparing RSA and ECDSA shows that when only the server is authenticated, RSA performs better, but when also the client is authenticated, RSA becomes less efficient than ECDSA. This is due to the asymmetry in RSA signatures, where signing is much more costly than verification. For ECDSA, these costs are much more symmetric.

Comparing the elliptic curves, the increase in energy when increasing the size is relatively constant, except for the case of RSA signatures. This suggests that using 4096-bit signatures is much more costly than using 2048-bit signatures. For high security levels and mutual authentication, ECDSA is thus highly preferred. However, without client authentication, RSA can be considered.

DHE (2048-bit prime) with RSA signatures requires much energy. The main reason is that this (Diffie-Hellman) uses an exponentiation with a secret value. It is thus clear that ECDHE should always be preferred over DHE.

Looking at the time for the handshake, it is often very slow. Analyzing the communication using Wireshark, we find that the vast majority of the time is being spent while waiting for the client to respond with the Client Key Exchange message.

Comparing the handshake to sending data, it costs around 6 μWh to send up to 1 KiB data, while any (non-PSK) key exchange will cost in the order of 50-500 μWh depending on cipher suite. Thus, minimizing the number of handshakes, and to make them more efficient, should be prioritized.

Key takeaways:

- Due to the asymmetric cost for signing and verifying RSA signatures, RSA is a viable option when client authentication is not used.
- ECDHE should always be preferred over DHE.
- Client computations are responsible for most of time and energy. This is particularly evident when the client computes RSA signatures, as is the case when we have mutual authentication (and RSA).
- Plain PSK is significantly more efficient than any other alternative.

Table 1: The handshake energy consumption (with standard deviation from 10 measurements is given in parentheses)

	Mutual authentication		One way authentication	
Cipher suite	CoAP energy consumption (μWh)	CoAP time (ms)	CoAP energy consumption (μWh)	CoAP time (ms)
TLS_DHE_PSK_WITH_AES_128_GCM_SHA256	268.66 (5.13)	4186 (23.06)	-	-
TLS_PSK_WITH_AES_128_GCM_SHA256	18.53 (0.61)	163.39 (5.28)	-	-
TLS_ECDHE(224r1)_RSA_AES_128_GCM_SHA256	114.33 (2.88)	1606 (8.14)	48.90 (3.90)	578.96 (34.78)
TLS_ECDHE(256r1)_RSA_AES_128_GCM_SHA256	125.00 (2.00)	1771 (12.16)	57.36 (1.59)	720.66 (9.44)
TLS_ECDHE(384r1)_RSA(4096)_AES_256_GCM_SHA384	463.66 (19.29)	7572 (367.73)	118.37 (20.09)	1678 (326.05)
TLS_ECDHE(521r1)_RSA(4096)_AES_256_GCM_SHA384	473.00 (15.39)	7691 (205.14)	155.00 (4.35)	2209 (112.42)
TLS_DHE_RSA_AES_128_GCM_SHA256	339.00 (1.00)	5395 (30.31)	276.33 (3.51)	4328 (19.15)
TLS_ECDHE_ECDSA(224r1)_AES_128_GCM_SHA256	106.33 (12.22)	1556 (217.41)	75.86 (0.90)	1033 (18.71)
TLS_ECDHE_ECDSA(256r1)_AES_128_GCM_SHA256	135.66 (0.57)	2098 (26.57)	100.23 (1.53)	1475 (7.63)
TLS_ECDHE_ECDSA(384r1)_AES_256_GCM_SHA384	178.00 (8.18)	2721 (151.71)	143.00 (2.64)	2168 (36.17)
TLS_ECDHE_ECDSA(521r1)_AES_256_GCM_SHA384	276.66 (7.37)	4432 (148.67)	213.33 (0.57)	3327 (26.40)

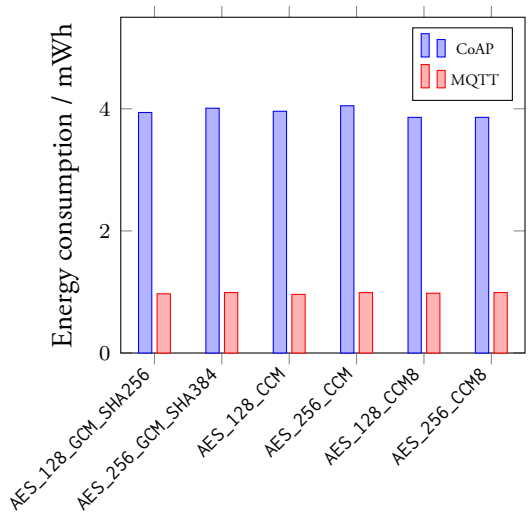


Figure 4: OTA firmware update energy consumption in CoAP and MQTT protocols with different cipher suites

6.3 Analysis of Firmware Update

In our experiments, we used the block-wise option in CoAP for transferring a firmware. Being around 870 KiB, it was transferred in 870 consecutive blocks from the server to the client, and then written to flash. We used MQTT as-is, since it handles the data size without problems. Upon completion, the client verifies the firmware and reboots into the new firmware.

In the experiments, we measured the energy between the client receiving the first and last firmware block. The results of the energy consumption, and also the time for receiving all the blocks, are shown in Fig. 4 and 5. As illustrated in Fig. 4, there is not much difference between different cipher suites in case of energy consumption for both CoAP and MQTT. The small differences that we can see in Fig. 4 are consistent with the previous measurement in that CCM8 is slightly more efficient than the other options (the two rightmost bars). Calculating a 95% confidence interval supports this observation. The time to receive all the encrypted firmware blocks in CoAP using CCM8 is also less than the time for other ciphers, as illustrated in Fig. 5.

We again notice that CoAP requires more energy and time compared to MQTT. This is consistent with the previous measurement shown in Fig. 2 and 3.

Key takeaway:

- For transferring the OTA update, MQTT is more efficient with a factor 4 for energy, and almost a factor 2 for time. This is because in CoAP, the firmware is transferred in several small blocks.

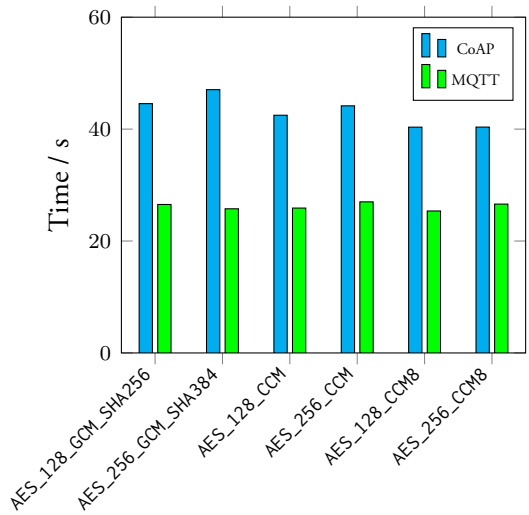


Figure 5: OTA firmware update duration time in CoAP and MQTT protocols with different cipher suites

6.4 Analysis of Packet Loss

We simulated packet loss using the traffic control utility `tc`[4] along with the network emulator `NetEm`[5] on the server side for both CoAP and MQTT with and without security. `NetEm` was configured to drop packets from the interface with a given probability. In our experiments, 500 packets with a fixed size of 512 bytes were sent from the client to the server (without any delay between sending the packets), with a loss rate up to 20 %. The energy consumption was measured to capture the effect of retransmission.

In bulk CoAP transmission, the packets will be sent consecutively until a packet loss occurs. When a loss occurs, the client waits until the next retransmission time. In the retransmission, the lost packet and also the remaining packets will be sent to the server.

Bulk transmission is handled differently in MQTT. Since MQTT utilizes TCP with its sender window, the sender might send the last packet while it is still waiting for previous packet acknowledgements. As a result, bulk transmission in MQTT with packet loss is much faster than CoAP since it does not have to wait for every packet to be acknowledged.

Since there was not much difference between key sizes and cipher suites, as discussed in Section 6.1, in these measurements we only consider AES with key size 128. The energy consumption for different packet loss rates for transferring 500 packets of size 512 bytes for CoAP and MQTT protocols are given in Fig. 6

[4]https://man7.org/linux/man-pages/man8/tc.8.html
[5]https://man7.org/linux/man-pages/man8/tc-netem.8.html

and 7. The energy consumption will increase with increasing packet loss, but this increase in CoAP, as illustrated in Fig. 6, is much more than in MQTT (note the different scaling of the y-axis). This is due to the differences in handling bulk transmission as explained above.

In Fig. 6 and 7, there is no significant difference between different modes of operation. NULL has the lowest energy consumption in both CoAP and MQTT, but in CoAP, the difference between NULL and other cipher suites decreases for loss rates higher than 10 percent. Because, in CoAP, when the rate increases, the waiting time between retransmissions also increases and the actual number of retransmissions decreases. The required energy for each second of waiting time between retransmissions is around 30 μWh. Sending the same number of packets (500 packets) in CoAP and MQTT, the required time in CoAP for 20 percent packet loss is around 7 minutes while in MQTT this time is only around 1 minute. This results in the huge difference between energy consumption in CoAP and MQTT protocols for higher packet loss rates.

We only consider a uniform distribution of the lost packages. Other packet loss distributions, e.g., burst errors, would likely affect the TCP window differently. Such specific network conditions should be take into account when analyzing the performance in case of packet loss.

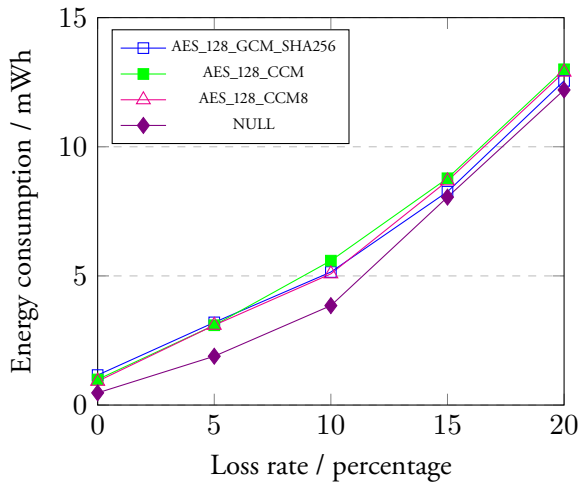


Figure 6: Energy consumption for CoAP/CoAPs protocols with different packet loss rates

The effect of the `ACK_TIMEOUT` parameter under different loss rates is shown in Table 2. Since much energy is consumed while waiting for the next packet, lowering waiting time can reduce energy. Comparing a 1 second `ACK_TIMEOUT` with the default 2 second value indicates that the control parameters can have significant impact on the energy consumption, in particular with high packet loss rates. As seen in Table 2, the energy consumption for 15 and 20% packet loss rates

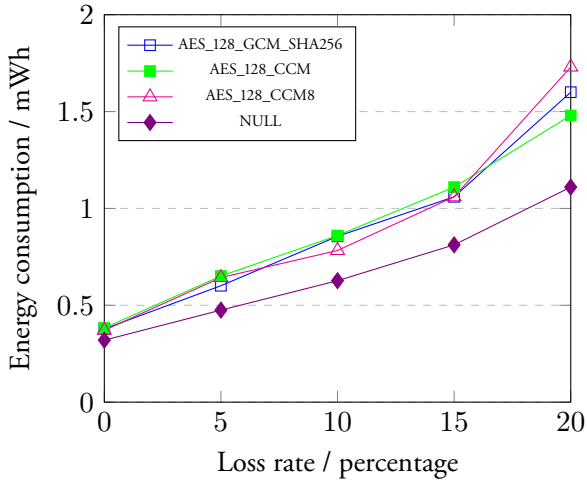


Figure 7: Energy consumption for MQTT/MQTTs protocols with different packet loss rates

with 2 seconds `ACK_TIMEOUT` is almost double the required energy for 1 second.

The experiments showed that by increasing the `ACK_TIMEOUT`, the energy consumption will be increased but this increase is more visible in higher packet loss rates. Therefore, it is important to optimize the retransmission parameters in CoAP protocol according to the environment.

Key takeaways:

- For CoAP on lossy networks, the encryption overhead decreases with the loss rate.
- MQTT performs significantly better in lossy networks due to the algorithms used in TCP.
- In CoAP, retransmission parameters such as `ACK_TIMEOUT` should be optimized according to the environment.

7 Conclusions

In this paper we give a better understanding of the difference in energy overhead for two common IoT protocols, CoAP or MQTT. We have done real-world experiments to measure and analyze the energy overhead of adding security to CoAP and MQTT on an ESP32 IoT device. During data transfer, we show that there is a constant energy overhead for up to around 1 KiB of data, which is the cost for setting up the WiFi connection. Above 1 KiB, CoAP has a higher penalty compared to MQTT, due to the block-wise transfer mechanism. Thus, for smaller

Table 2: The effect of ACK_TIMEOUT parameter on CoAP protocol energy consumption

	Energy Consumption (mWh)				
ACK_TIMEOUT (s)	1	1.5	2	2.5	3
5% packet loss	1.39	1.55	1.88	2.38	2.50
10% packet loss	2.39	3.16	3.85	4.85	6.29
15% packet loss	3.98	6.00	8.05	9.86	10.73
20% packet loss	6.46	7.78	12.20	13.58	16.65

packet sizes, CoAP and MQTT are comparable, while MQTT is favorable for larger packets. Due to the overhead of setting up the wireless connection, it is clear that data should be aggregated, whenever possible.

Looking at the DTLS handshake, ECDHE should always be preferred over DHE, while RSA as digital signature has a slight advantage when client authentication is not needed.

In most cases, MQTT outperforms CoAP, much due to the window based retransmission strategy in the underlying TCP protocol. The most suitable use case for CoAP is on a reliable network, transmitting small packets of data. For other cases, the strength of TCP becomes evident.

Note that the numbers provided are hardware and implementation specific. Also there are other factors that can affect the results, such as environmental conditions, network topology, integration of many sensors in IoT systems, device use case, etc. As a result, all of above factors need to be considered in the energy consumption of IoT devices.

Acknowledgments

This work was supported in part by the Swedish Foundation for Strategic Research, grant RIT17-0035 and by the Wallenberg AI, Autonomous Systems and Software Program (WASP).

References

[Bar19] E. Barker. “Draft NIST Special Publication 800-57, Part 1, Revision 5”. In: *National Institute of Standards and Technology (NIST)* (2019).

[Bor18] C. Bormann. *CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets*. RFC 8323. https://tools.ietf.org/html/rfc8323. RFC Editor, Feb. 2018.

[BZ16] C. Bormann and E. Z. Shelby. *Block-Wise Transfers in the Constrained Application Protocol (CoAP)*. RFC 7959. https://tools.ietf.org/html/rfc7959. RFC Editor, Aug. 2016.

[Cap+15] A. Capossele et al. "Security as a CoAP resource: an optimized DTLS implementation for the IoT". In: *2015 IEEE international conference on communications (ICC)*. IEEE. 2015, pp. 549–554.

[Esp19] Espressif Systems. *ESP32-WROOM-32D & ESP32-WROOM-32U*. https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32d_esp32-wroom-32u_datasheet_en.pdf, Last accessed 2019-05-29. 2019.

[Esp20] Espressif. *Espressif IoT Development Framework*. https://github.com/espressif/esp-idf, Last accessed 2020-01-20. 2020.

[GMS15] J. Granjal, E. Monteiro, and J. S. Silva. "Security for the internet of things: a survey of existing protocols and open research issues". In: *IEEE Communications Surveys & Tutorials* 17.3 (2015), pp. 1294–1312.

[IOU17] M. Iglesias-Urkia, A. Orive, and A. Urbieta. "Analysis of CoAP implementations for industrial Internet of Things: A survey". In: *Procedia Computer Science* 109 (2017), pp. 188–195.

[MC19] K. McKay and D. Cooper. *Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations*. Tech. rep. National Institute of Standards and Technology, 2019.

[Raz+13] S. Raza et al. "Lithe: Lightweight secure CoAP for the internet of things". In: *IEEE Sensors Journal* 13.10 (2013), pp. 3711–3720.

[Sel+19a] G. Selander et al. *Object Security for Constrained RESTful Environments (OSCORE)*. RFC 8613. RFC Editor, July 2019.

[SHB14a] Z. Shelby, K. Hartke, and C. Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. http://www.rfc-editor.org/rfc/rfc7252.txt. RFC Editor, June 2014.

[Suá+18] M. Suárez-Albela et al. "A Practical Performance Comparison of ECC and RSA for Resource-Constrained IoT Devices". In: *2018 Global Internet of Things Summit (GIoTS)*. IEEE. 2018, pp. 1–6.

[TBK19] A. Thantharate, C. Beard, and P. Kankariya. "CoAP and MQTT Based Models to Deliver Software and Security Updates to IoT Devices over the Air". In: *2019 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)*. IEEE. 2019, pp. 1065–1070.

[Tha+14] D. Thangavel et al. "Performance evaluation of MQTT and CoAP via a common middleware". In: *2014 IEEE ninth international conference on intelligent sensors, sensor networks and information processing (ISSNIP)*. IEEE. 2014, pp. 1–6.

RoSym: Robust Symmetric Key Based IoT Software Upgrade Over-the-Air

Abstract

Internet of Things (IoT) firmware upgrade has turned out to be a challenging task with respect to security. While Over-The-Air (OTA) software upgrade possibility is an essential feature to achieve security, it is also most sensitive to attacks and lots of different firmware upgrade attacks have been presented in the literature. Several security solutions exist to tackle these problems. We observe though that most prior art solutions are public key-based, they are not flexible with respect to firmware image distribution principles and it is challenging to make a design with good Denial-Of-Service (DoS) attacks resistance. Apart from often being rather resource demanding, a limitation with current public key-based solutions is that they are not quantum computer resistant. Hence, in this paper, we take a new look into the firmware upgrade problem and propose RoSym, a secure, firmware distribution principle agnostic, and DoS protected upgrade mechanism purely based on symmetric cryptography. We present an experimental evaluation on a real testbed environment for the scheme. The results show that the scheme is efficient in comparison to other state of the art solutions. We also make a formal security verification of RoSym showing that it is robust against different attacks.

Pegah Nikbakht Bideh and Christian Gehrmann. "RoSym: Robust Symmetric Key Based IoT Software Upgrade Over-the-Air". In *the 3rd Workshop on CPS&IOT Security and Privacy (CPSIOTSEC 2022), Los Angeles, U.S.A.*

1 Introduction

The Internet of Things (IoT) is a common term for describing systems of interconnected devices. The devices can be of many different types and are used in divergent local networks and with a wide range of capabilities. Some units are very powerful while others are extremely tiny and resource constraint with respect to computing capabilities, volatile and non-volatile memory sizes, etc. IoT devices are not directly human operated, they are often managed remotely [Fer+17]. This means that software updates and other critical maintenance operations need to be performed over the network and when the device is wireless, often referred to as Over-The-Air (OTA) updates. Several severe attacks against IoT Firmware upgrades have been reported in recent years. The attacks are of different types either attacking the firmware during transfer [Shi+17] or through Denial of Service (DoS) of the actual update process [Fan+21]. Hence, there are very good reasons to offer highly secure and robust software upgrades for IoT systems. As we discuss in Section 2, lots of efforts have historically been put into code dissemination solutions for Wireless Sensor Networks (WSNs) [BS13]. Even if wireless IoT networks share many WSN characteristics, they also have some specific characteristics and needs. Especially, IoT firmware upgrade does not always happen through direct multicast, but through many other distribution mechanisms [Ara+21]. This put special requirements on how the upgrade packages are verified and protected. In particular, the firmware upgrade scheme must in such a situation be able to handle both out of order delivery and intermediate storage of upgrade packages. Another problem is that most existing firmware upgrade solutions are public key-based using non post-quantum resistant algorithms. Even if it would be possible to transfer existing public key schemes into ones using quantum cryptographic algorithms, those currently available do not promise efficient enough signature algorithms in terms of performance and size compare to currently non post-quantum resistant algorithms [Beu21]. This is a problem when we consider resource constraint IoT units. On the other hand, to resist Grover's algorithm [Gro96b] against symmetric key primitives, only a double of the key size is needed. Consequently, there is a need for investigating new, completely symmetric key-based upgrade solutions.

Our new scheme, RoSym, is a new solution addressing exactly these concerns. We have looked into the IoT firmware upgrade problem with a focus on the requirement of being transport agnostic while also being able to handle the most recent DoS threats against the upgrade procedure. We have worked with a design offering post-quantum resistance by limiting the solution to only being dependent on pure symmetric key operations. RoSym is an upgrade scheme that has very low complexity. The design approach has been very pragmatic, taking state-of-the-art symmetric cryptography and well-proven techniques and bringing them together to make an overall firmware upgrade solution that can easily be implemented, both on the IoT and the software management side of the system. The main contributions of the paper are the following:

- We consider the secure IoT software upgrade problem and identify the main security, robustness, and performance requirements for software distribution transport agnostic, pure symmetric key protected upgrade.
- We propose a new transport agnostic and secure software upgrade scheme with good DoS robustness fulfilling the identified requirements.
- We implement the proposed solution and present performance figures.
- We make a formal security verification of the proposed protected upgrade scheme.

The rest of this paper is organized as follows, in Section 2, prior art firmware upgrade techniques for WSNs and IoT networks are reviewed. Section 3 gives problem background and derives main design requirements. In Section 4 and Section 4, the design of our new scheme is given. We describe our implementation in Section 6. The evaluation results are represented in Section 7. The formal security verification of our scheme is given in Section 8 and finally, the paper is concluded in Section 9.

2 Related Work

Next, we present the most relevant related work. We start by discussing different code dissemination protection principles previously suggested in the literature. We then treat the DoS aspect of firmware upgrade in the related art. A comparison of prior-art solutions with respect to these aspects is given at the end of the section.

2.1 Secure Code Dissemination Approaches

The problem of software update or code dissemination has been studied in the area of WSN and IoT for a long time. One of the first papers discussing the code dissemination problem for WSN was the pioneering paper on the scheme Deluge [HC04]. Deluge did only treat the distribution principle as such, suggesting that a firmware image is divided into fixed size pages and then pages are divided into fixed size packets, which are then distributed to the networked devices. However, the security aspect was never treated in the original Deluge scheme.

Later, many secure versions of Deluge were proposed. For example, in secure Deluge [Dut+06], after the division of code image to packets, a hash is computed over the last packet and is appended to the end of the previous packet and similar hashes are embedded recursively to all packets. Then, the base station signs the first hash and the receiver verifies the signature and stores the hash to authenticate the next packet. Sluice [LGN06] is similar to secure Deluge but in Sluice page level hashes are used instead of packet level hashes. Other Deluge protection solutions were at about the same time proposed. The authors in [DHM06], investigate

different trade-offs between a special hash tree structure with signed roots and a hash chain. A rather similar approach is taken by the solution in [Hyu+08], called Seluge. Seluge builds a Merkle hash tree with hash values of packets on the first page, the rest of the packets can then be authenticated by a hash chain. All these previous methods are public key based. Even if Seluge uses conventional public key encryption (non post-quantum resistance), it is interesting to compare Seluge with our approach and we present performance figures in Section 7. Several other public key based protocols have been proposed such as SDRP [He+11], SCATTER [KD11], SenSeOP [Asc+12] and Flexicast [LKK15].

Similar to our solution, also different pure symmetric key based techniques have been introduced. PETRA [IKC09] is based on symmetric keys, and its security is built upon a combination of Message Authentication Codes (MACs) and a Bloom-filter technique. One MAC is used to protect the Bloom-filter and another to verify the whole software image. The drawback of the Bloom-filter is that it gives false positive answers which might not be acceptable in some circumstances. Similar to our approach, PETRA assumes a common MAC key among all devices.

Castor [KGN07], another symmetric scheme, similar to [DHM06] uses a one-way hash function to verify the software update packages. This approach shares the characteristics of our scheme with respect to the usage of a hash chain with a root value. However, it does not suggest any individual packet verification making it more vulnerable to battery drain attempts and it uses a much less practical distribution principle of the root hash value. Furthermore, packet confidentiality is not considered in Castor. We compare our solution to PETRA and Castor as well. μTESLA [Per+02a] is yet another generic multicast data distribution scheme that uses a one-way key chain and a delayed key method applicable also to code dissemination. μTESLA is very efficient since it purely uses symmetric cryptography but does not give very strong integrity guarantees and requires time synchronization between the distributing unit and the IoT units.

Also, different combinations of asymmetric and symmetric key approaches exist. For instance, SECNRCC [Xie+15] combines a hash tree and key chain to provide confidentiality and authentication. Similar to other solutions, in SECNRCC, the hash values in a Merkle tree and the root hash are signed by the software distributor. In SECNRCC, the packets with hash values are also encrypted with a session key. DoS resistance is provided through the usage of a special purpose symmetric key chain with individual keys distributed to all units.

Later, IoT oriented (instead of WSN) firmware upgrade procedures have been suggested. SEDA [Kim+16] is one such multicast-based update scheme for IoT environments. In SEDA, a secure group key distribution mechanism is used which requires pre-installation of public and private keys on all IoT units using classical public key algorithms. The evaluation in [Kim+16] indicates better performance compare to methods such as Seluge [Hyu+08]. Hence, we find it relevant to compare our approach with SEDA as well and the figures are presented in Section

7. ASSURED [Aso+18] is another OTA firmware upgrade solution for embedded devices taking a life-cycle perspective on the software upgrade by considering four different system entities: 1) an original equipment manufacturer (OEM), 2) a firmware distributor, 3) a domain controller and 4) a connected device. In the ASSURED approach, the OEM signs the new firmware using ECC and the devices need to perform ECC signature verification before installing the firmware. In the adopted model, OEM and domain controller keys need to be stored on devices at manufacturing time. This gives some additional complexity to the realization of the solution.

All the previously discussed approaches are quite complicated and have a performance impact due to the fact that they integrate the verification protection under the assumption that the firmware distribution must be done in a direct fashion (using for instance multicast). In all these approaches when a hash or chain of hashes is used for verification of the firmware blocks, the whole process starts or integrates the secure distribution of these hashes into the firmware dissemination protocol. In this paper, we take a much more practical and pragmatic approach by *dividing* the firmware upgrade into *two* phases, in one phase very essential integrity data (session key material and hash value) are done *prior* to the phase of actual distribution or download of the actual image. Such an approach makes it possible to make the actual firmware loading process *independent* of the transport method, i.e., it allows any suitable distribution mechanism of the software packages. This is inline with how most modern IoT units are connected to a back-end management system. This is also a much more feasible principle to use than for instance a most recent suggestion to use an injection of integrity checks, directly into the firmware [VRM21] as part of the software distribution.

2.2 DoS Protection Aspects

The possibility for an attacker to use the software dissemination mechanism to issue DoS depends on how the actual firmware packages are protected during transfer and when processed on the IoT unit. If the whole image must first be installed *before* the integrity can be checked, this constitutes a major security risk. This can take for instance the form of some type of "repacking" as reported in [Shi+17] demonstrating a practical attack on commercial fitness trackers. Just the fact that large parts of the firmware image must be processed before the validity can be checked constitutes a DoS risk such as a battery drain attempt against a battery driven IoT unit. This is for instance the case for the previously discussed schemes Secure Deluge and Sluice. This type of simple DoS aspect has been tackled by many of the later schemes such as Seluge [Hyu+08] using a Merkle tree approach or the SECNRCC scheme [Xie+15], which combines a key chain with a hash chain to achieve the same thing. Another approach with the same aim is the scheme presented in [Tan+13], which also uses Deluge as the basic distribution scheme. It provides confidentiality by leveraging session keys derived from

hashed data packets. In this scheme, a *Cipher Puzzle* is used as a weak authenticator for DoS protection. It is here worth noticing that a *Cipher Puzzle* can cause sender-side delay which might not be acceptable in every network.

IoT units are also sensitive to direct physical tampering. This can be used to attack the scheme by for instance using power analysis [Ron+17]. Countermeasures to handle this are to use IoT unit individual keys and/or public key protection of the firmware upgrade. In the paper by Yan-Hon Fan et al., [Fan+21], it is noticed that these approaches are not feasible for many resource constraint devices and they instead suggest a limit to the number of allowed upgrades per 24 hours to five. This will prevent the power analysis attack described in [Ron+17] but it also opens up for DoS attacks by the attacker managing to fill up the number of updates to the maximum allowed number. The authors in [Fan+21] suggest an application layer protocol between the update server and the IoT unit to solve this issue. In this paper, as discussed above, we use a similar approach by dividing the update process into two phases, one phase handles the key material and the second one handles the actual upgrade. Different from the solution in [Fan+21], we suggest a trade-off between individual packet MAC checks combined with a more traditional hash chain. The advantage of this is to have better protection against the previously discussed packet modifications [Shi+17], i.e. DoS attacks during the actual firmware load.

A completely different way of handling the upgrade DoS problem, which typically occurs over a wireless interface, is to instead focus on detecting them. Such solutions have been reported in several papers [JHC22; SS21], but that is an orthogonal problem and solution to the prevention mechanisms and can be used in parallel with our solution.

2.3 Protected Code Dissemination Comparison

We compare the different state of the art solutions for code dissemination in WSN or IoT networks in Table 1. Table 1 summarizes the characteristics of the reviewed schemes in Section 2 and our scheme includes used cryptography, DoS protection, and source authentication mechanisms. In Table 1, the schemes which use both symmetric and asymmetric cryptography are marked as mixed. Table 1 indicates that in most asymmetric schemes, Merkle hash tree and digital signatures are used to authenticate the messages while in symmetric schemes, including our scheme different MAC verification techniques are used instead. Different DoS protection methods including Puzzle based approaches are used but Puzzle based approaches have some drawbacks which will be further discussed in Section 4. The main difference between our scheme with other symmetric approaches is the independence of the distribution mechanism (multicast/broadcast, direct download, etc.) and also the use of time limited MAC verification for DoS protection.

Table 1: Comparison of existing code dissemination procedures and our scheme

Features	Cryptography	DoS resistance	Source authentication	Multicast
Deluge [HC04]	No security			
Secure Deluge [Dut+06]	Asymmetric		Digital signatures and packet level hash	
Sluice [LGN06]	Asymmetric		Digital signatures and hash chains	
[DHM06]	Asymmetric	Signed hash tree verification	Digital signatures	
Seluge [Hyu+08]	Mixed	Immediate authentication, Message Specific Puzzle	Merkle hash tree and digital signatures	
SDRP [He+11]	Asymmetric		Merkle hash tree and digital signatures	
DiCode [He+12]	Asymmetric	Immediate authentication, Message Specific Puzzle	Merkle hash tree and digital signatures	
SCATTER [KD11]	Asymmetric	Signature verification	Merkle hash tree and r-time signatures	
SenSeOP [Asc+12]	Asymmetric	Signature verification and time lock	Hash value and digital signatures	Yes
SECNRCC[Xie+15]	Mixed	Cipher Puzzle	Digital signatures	
Felxicast [LKK15]	Asymmetric		Fingerprint with bloom filter	
[Tan+13]	Mixed	Cipher Puzzle	Digital signatures	
PETRA [IKC09]	Symmetric		MAC with bloom-filter	
Castor[KGN07]	Symmetric		MAC with hash chain	
μTESLA [Per+02a]	Symmetric		MAC with time synchronization	
SEDA [Kim+16]	Mixed	HMAC verification	HMAC and Digital signatures (Advertisement packets)	Yes
ASSURED [Aso+18]	Asymmetric		Digital signatures	
RoSym	Symmetric	Time limited MAC verification	MAC with hash chain	Yes

3 Problem Definition and Requirements

We consider a threat model where an attacker is able to interfere with any part of the software distribution chain including any potential intermediate storage of the whole or part of the firmware image. This means that an attacker has the power to modify and read upgrade packages or interfere with any communication to and from the IoT unit. However, we assume the IoT management system, or what we refer to as the Device Management System (DMS) to be fully trusted not under the control of an attacker. When it comes to the IoT units themselves, we adopt a trust model where attacks on a single or a few IoT units are possible but typically time and resources consuming as is the case for direct tampering of the unit or if the attacker use for instance a power analysis to get access to long or short terms keys [Ron+17]. We consider it infeasible for the attacker to get direct control of a large number of the deployed IoT units to perform such an attack.

In Section 2, we discussed several previous design efforts for secure and robust software upgrades of wireless units. Many solutions use a reverse hash chain delivery or Merkle hash tree for the integrity protection of the software upgrade following the threat model above completely or in part. However, as we concluded in the review, the existing solutions typically have one or several of the following drawbacks in a resource constraint IoT setting:

- Depends on non post-quantum resistant public key operation or operations on the IoT side,
- Cannot handle out of order packet delivery,
- Do not offer individual checks of packets making the scheme vulnerable to battery drain attacks,
- Require complex Puzzle solving or time synchronization on the IoT side,
- Require direct multicast or broadcast delivery not supporting intermediate storage of upgrade data.

We conclude that a secure and robust software upgrade scheme of resource constraint units using pure symmetric key should fulfill the following requirements:

R1. **Integrity and confidentiality protection**: The integrity and confidentiality of individual software packets, as well as the complete software distribution, must be guaranteed.

R2. **DoS robustness**: It must not be possible for an attacker to use false software packet distribution to force the IoT unit to consume significant computing, power, and/or memory resources on a single or several IoT units.

R3. **Efficient communication and computational cost**: The software upgrade process shall require as little bandwidth and resources as possible.

R4. **Efficient memory requirements**: The software upgrade process shall use as little IoT volatile and non-volatile memory resources as possible.

R5. **Transport agnostic upgrade**: The upgrade scheme shall support direct download from local or remote update servers as well as direct or local multicast delivery and out of order delivery of upgrade packages. The scheme shall allow intermediate storage of the upgrade images.

In this paper, we seek a protected, robust, and secure code dissemination scheme that fulfills all these requirements.

3.1 Notations

For the rest of the paper we use the following notations:

- Denote the set of IoT units subject to upgrade by $U = \{u_0, u_1, \ldots, u_{w-1}\}$.
- In our scheme, IoT units are controlled by a back-end system, or, what we refer to as the DMS.
- We denote the complete new software upgrade information by I = I_0, I_1, I_2,..., I_{n-1}, i.e. the upgrade information is split into n distinct pieces.
- The software image is distributed using a sequence of software packages denoted by P = P_0, P_1, P_2,..., P_{n-1}. These upgrade packages not only hold software image parts, I, but also additional information.
- $\forall u \in U$, a shared long-term secret between u and DMS is denoted by K_u.
- An integrity protection key used by the DMS to protect the integrity of software upgrade packages is denoted by IK_{sw}.

- A confidentiality protection key used by the DMS to encrypt the software packages prior to distribution is denoted by K_{sw}.
- T_1 is a first time parameter set by the DMS that indicates the validity period of each upgrade packet.
- T_2 is a second time parameter defining the validity of IK_{sw} and K_{sw}.
- h_i is a one-way secure hash and denotes hash of P_i.
- $C_i = E_{K_{sw}}(I_i)$ is an encrypted software update block.
- We assume each software package is integrity protected using a MAC denoted by $MAC_i = MAC_{IK_{sw}}(D_i)$, where D_i is a protected subfield of P_i.

4 Design Features of RoSym

Before describing the details of RoSym, here, first, we briefly explain the communication and security features of our scheme. As previously discussed, the basic assumption in our solution is the possibility to distribute key material and firmware image hash values *prior* to the actual firmware download process. This is different from the prior-art solutions but has the advantage that we both achieve transport agnostic download and very low complexity with respect to security checks and distribution format. This requires that the IoT unit has a security relationship and performs a handshake with a trusted entity responsible for the code distribution, i.e. the DMS. While one might think that this limit the applicability of the design, we instead argue that this is indeed inline with state of the art IoT architecture, which typically is under the control of a cloud-based management system. Examples of such systems are Thingsboard[1] and Mainflux[2]. Below, we further discuss the precise design assumptions of RoSym.

Code dissemination principles: We assume a central unit, DMS, to be responsible for the code dissemination and code protection preparations. Each IoT unit must have direct contact with DMS prior to software installation. The DMS distributes the actual code image according to the choice best suited for the particular IoT scenario. A central or several local servers can be used for direct download, packet by packet by the IoT unit, or a multicast or unicast protocol can be used for the actual firmware distribution, i.e. our solution is completely firmware transport agnostic.

Reduced communication load and energy consumption: Energy consumption of IoT units can be significantly decreased by reducing their active time. They can be programmed in a way to wake up in defined time intervals to send alive

[1] https://thingsboard.io/
[2] https://www.mainflux.com/

packets. As our solution allows arbitrary code distribution, the distribution server can wait until a unit becomes alive to trigger a new upgrade.

Integrity and confidentiality protection: In RoSym, we used MAC to protect the integrity of update packages. For that, the DMS and IoT units need to setup a secure key prior to the update procedure. Although symmetric cryptography requires lower computational resources, if a single device gets compromised, the adversary can compromise the whole procedure. As a result, any time a group of IoT units needs to be updated, integrity and confidentiality protection keys, IK_{sw} and K_{sw}, are transferred to the units subject to the software update, then the transferred keys are stored securely on the IoT units. These keys will be revoked or expired at the end of the upgrade procedure.

DoS protection: DoS attacks exploiting authentication delays are important to address [Hyu+08]. There are a number of broadcast weak authenticator mechanisms for symmetric encryption including *Message Specific Puzzle* (MSP) [Hyu+08; He+12] and *Cipher Puzzle* (CP) [Tan+13] which can be used. Although, these mechanisms have high security they can cause unreasonable sender-side delays. In order to decrease this delay, a *Dynamic Cipher Puzzle* (DCP) method was proposed in [AS+18]. This method decreases the sender-side delay but the used resources, consumption time, energy and RAM will be increased on the receiver units. In our scheme due to the mentioned limitations, we introduce time limited MAC verification, instead of these techniques. One way of deploying a time limit in the MAC verification is to time synchronize all of the IoT units in the network. A variety of secure time synchronization approaches have been proposed for WSN or IoT networks [San07; Qiu+17]. Although these methods are valuable they increase the complexity, overhead, and energy consumption of IoT units. Instead of synchronizing the IoT units, we keep local packet arrival times on all IoT units and only allow a maximum T_1 delay between individual packets and a maximum of T_2 for the total upgrade time, i.e. the time from that the IoT unit received the keys IK_{sw} and K_{sw} until the upgrade must be finalized. When different packets arrive, the arrival times will be recorded by the IoT units based on their local time and then the time difference between different packet arrivals will be considered, as a result, time synchronization is not needed. In order to prevent other types of DoS attacks including jamming attacks, other prevention mechanisms can be used in parallel with our solution.

5 Solution

The solution we propose, is as previously discussed, based on the principle that each IoT unit has a trusting relationship with the DMS through the sharing of an IoT unique, long-term secret. This secret is used by the DMS to keep control of all IoT units in the system and when a firmware upgrade is about to take place, securely transfer upgrade key material to all the IoT units. Once this is done, the DMS can use the code dissemination channel for the actual firmware download.

Here, to defend against DoS attacks, as discussed in Section 4 above, we use a limited time window, i.e. an update must take place within a certain time period. This is true both with respect to the maximum time allowed for the delay of two consecutive download packages to arrive and the total time allowed for an upgrade. Once this period has elapsed, the keys used to protect the firmware are no longer valid and will be refused by the IoT units. In this section, we described the detailed procedures of the needed steps.

5.1 Key Establishment and Parameter Setup

We assume that on the first setup of the IoT units, the DMS URI/s (based on the number of available DMS server/s) will be included in all IoT units. A long-term shared key or K_u is stored on each unit and the DMS server/s as well. The IoT units send with some regularity alive messages to the "owning" DMS server on the network. In case of available updates, the response from the server includes update availability, the number of seconds to wake up after receiving the response, and a wake up time window to be awake during that time (to avoid time synchronization requirements). If they get a response that indicates the availability of a new software upgrade, they will exchange upgrade parameters with the DMS on the specified wake up time window as described below.

During the wake up time window of IoT units and before the units receive the new software upgrade image, a secure configuration session needs to be setup between the DMS and the IoT units. Any appropriate secure protocol can be used but in this paper, the Object Security for Constrained RESTful Environments (OSCORE) protocol [G S19] is used to protect the transfer using K_u as a master secret. After configuration of the secure session, the DMS transfers the following information to all IoT units in the network or either the ones inside the multicast group:

1. Two randomly generated symmetric keys, IK_{sw} and K_{sw}.
2. Timing information, T_1 and T_2 determines the maximum allowed delay between two consecutive software packages and the validity of IK_{sw} and K_{sw} keys respectively.
3. A software image one-way hash root value, h_0.
4. The number of packages (*n*) in the new software distribution.

5.2 Upgrade Procedure

Before the upgrade procedure starts, the DMS also needs to package the software update image into suitable size packages. The package format is indicated in Figure 1 and the different parts will be explained in detail below. The related size

selections are given in Section 6.2. According to our solution, all software packages are integrity and confidentiality protected. The purpose of this protection is not to give a very high protection level (as the keys need to be shared with a large number of devices), but above all, to make it harder for an attacker to perform a direct attack on single packets (DoS).

The different software packages can then be distributed to the IoT units in any way, i.e. the different packages can for instance be downloaded to several local software upgrade servers, where they are in turn fetched by the IoT units. They can also be distributed using multicast by the DMS to all the units. In our proof of concept implementation described in Section 6, this is the principle used. According to our design, the DMS performs the following steps:

1. The software update image is split into *n* distinct parts: I_0, I_1, I_2,..., I_{n-1}.

2. Calculate *n* different software upgrade packages:
   $P_i = \{i, E(K_{sw}, I_i), h_{i+1}, MAC_i\}$, $0 \leq i \leq n\text{-}2$,
   $P_{n-1} = \{n\text{-}1, E(K_{sw}, I_{n-1}), MAC_{n-1}\}$,
   where E() denotes the encryption using a suitable symmetric encryption scheme under the key K_{sw}[3] and where:

   (a) $h_i = H(\{i, E(K_{sw}, I_i), h_{i+1}\})$, $0 \leq i \leq n\text{-}2$,
   $h_{n-1} = H(\{n-1, E(K_{sw}, I_{n-1})\})$, and where H is a suitable, secure one-way hash function such as SHA-2 [PUB12] or SHA-3 [DRA14]. The hash of each package is dependent on the next package hash value.

   (b) $MAC_i = MAC(IK_{sw}, \{i, E(K_{sw}, I_i), h_{i+1}\})$, $0 \leq i \leq n\text{-}2$,
   $MAC_{n-1} = MAC(Ik_{sw}, \{n-1, E(K_{sw}, I_{n-1})\})$, where MAC denotes a suitable message authentication code function under IK_{sw}, such as HMAC [KBC97a] or short MAC [GTH15].

3. On the wake up time window of the IoT units, $\forall u \in U$ the following happens:

   (a) A confidentiality and integrity protected session based on the key K_u, is established between *u* and the DMS.

   (b) By using the secure session, the DMS transfers at least the following parameters to *u*: IK_{sw}, K_{sw}, h_0, T_1 and T_2.

5.3 Upgrade Procedure on IoT Unit Side

On the IoT side, the procedure starts with the actual firmware download credentials receiving and preparing as described in Section 5.1. Next, the IoT unit makes

[3] Typically this encrypted structure will also include a suitable IV.

Field name	index: 4B	Enc software: 944B	$hash_i$: 32B	MAC_i: 32B

index	Package index, *i*
Enc software	Encrypted software block: E(K_{sw}, I_i)
$hash_i$	One-way hash, h_i = H({i, E(K_{sw}, I_i), h_{i+1}}), 0 ≤ *i* ≤ *n-2*
MAC_i	Message Authentication Code, MAC_i = MAC(IK_{sw}, {i, E(K_{sw}, I_i), h_{i+1}}), 0 ≤ *i* ≤ *n-2*

Figure 1: RoSym software package format and size

preparations to receive the firmware image. During the image transfer, each software package is integrity and confidentiality protected to prevent both malicious code read and direct packet modification done with the purpose of for instance wasting IoT resources. Also, the IoT unit keeps track of time parameters to prevent DoS attacks. Here it is important to notice though that these clocks are *internal*, i.e. there is no need for any synchronization across the system. The detailed step-by-step procedure for receiving the actual software and installation is given below:

1. Get the time of arrival of the control parameters packet and store it as t_c.
2. Set *i = 0*,
3. Get the next software package P_i, and store its' arrival time as t_i.
4. If t_i - $t_{i-1} > T_1$, *1 ≤ i ≤ n-1*, the software upgrade is aborted and no more package is accepted by the IoT unit. If *i == 0*, consider t_c as t_{i-1}.
5. If t_i - $t_c > T_2$, *0 ≤ i ≤ n-1*, the software upgrade is aborted. Else, verify the integrity of P_i, by calculating MAC_i using the key IK_{sw}, over the fields *i*, *E(K_{sw}, I_i)*, h_{i+1}, if 0 ≤ *i ≤ n-2*, and over the fields *i* and *E(K_{sw}, I_i)*, if *i == n-1*.
6. Compare MAC_i with MAC'_i in the received package and only if coincide, accept the new package.
    - If 0 ≤ *i ≤ n-2*:
        (a) Verify the integrity of package P_i using the hash h_i (previously received and stored hash) and h'_i (calculated hash over i, $E(K_{sw}, I_i)$ and h_{i+1}). If the verification fails, request retransmission of P_i.
        (b) Store h_{i+1}, existed in the received package, in RAM memory. (Note: if h_i is not available in memory due to out of order packet delivery, only h_{i+1} will be stored in memory and verification of h_i will be postponed until previous package arrives.)

(c) Decrypt software upgrade package, I_i, using the key K_{sw} and store it in flash memory.

- Else:
  (a) Verify the integrity of software package P_{n-1} using h_{n-1} and h'_{n-1}. If the verification fails, request retransmission of P_{n-1} or try to fetch it again from a distribution server (if a direct download is applied).
  (b) If the verification is successful, decrypt the software upgrade package, I_{n-1}, using the key K_{sw} and store it in flash memory.

7. set $i = i+1$.
8. If $i < n$, repeat step 2.
9. Install the complete new valid software image, $I = I_0, I_1, I_2,..., I_{n-1}$.

Error Handling

The upgrade procedure of IoT unit/s (at any step) might be disturbed due to unexpected errors. If an error occurs before the update procedure starts, the DMS will realize this on receiving the next alive message and it will upgrade the failed units in a unicast way again. On the other hand, if the error occurs in the middle of the upgrade procedure, the failed unit will send a direct request to the DMS, and the procedure can be resumed from where it failed.

6 Implementation and Experiments

In our implementation of RoSym[4], we use a multicast distribution of firmware upgrade image directly from the DMS, and the UDP protocol is used as the underlying transport protocol. Our implementation consists of DMS and IoT unit realizations. IoT units communicate with the DMS over WLAN. All IoT units in the system send alive messages periodically to the DMS server. The alive messages include the current firmware version along with other information about the IoT unit. Once an IoT unit sends an alive message and receives an acknowledgment from the DMS, it goes to sleep until the next alive message needs to be sent. If there is an update available on the DMS, it piggybacks the number of seconds that IoT units need to wake up after receiving the acknowledgment, and the awake time window (this window can be decided by the administrator based on network delay) on the acknowledgment. Then, the IoT unit will be added to a multicast group by the DMS as well. No extra time synchronization is needed since by receiving the acknowledgment the IoT units calculate their wake up time based on

[4]Our implementation is available at: `https://github.com/pegahnikbakht/RoSym`

their internal local time and they prepare to wake up and be awake during the specified time window, this window is required since different devices may receive the acknowledgments with some delay.

After identifying the IoT units which need to be updated, the DMS generates IK_{sw} and K_{sw}. The DMS divides the plain firmware into different chunks with the size of 944 bytes (the size selection will be explained in Section 6.2) and each chunk will be encrypted using K_{sw}, then, the hash chain will be calculated. Finally, using the key IK_{sw}, the MAC of each chunk will be calculated using the encrypted value of the current chunk and the hash of the next chunk.

After preparing the secure firmware, on the wake up time window of the IoT units, a secure OSCORE session using K_u as the master secret will be established to the identified IoT units and then the randomly generated keys along with other required information will be sent to the units. Then, the secure session will be closed by the DMS. Immediately after that, a multicast socket will be opened on the target IoT units. Finally, those units will receive the upgrade multicasted packages and they will perform the verification, and after successful verification of all packages, the units will boot the new firmware. The information flow between IoT units and the DMS is shown in Figure 2.

6.1 Hardware Choice

In our implementation, we have used ESP32-S2[5], which is a single core board with Xtensa 32-bit LX7 CPU which operates at up to 240 MHz. ESP32-S2 is low power and single-core WiFi microcontroller, it is also cost-efficient, and has high performance with the following features:

- Support for cryptographic hardware accelerators to enhance performance
- Good protection against physical fault injection attacks
- Protection of private key and secrets from software access
- Integrating a set of peripherals, with different programmable GPIOs which can provide USB OTG, LCD interface, camera interface, UART, etc.
- It can be configured with both MbedTLS and WolfSSL libraries but in this paper, the default SSL/TLS library or MbedTLS is used.
- It has an official development framework[6] and we used it in our implementations as well.

[5] https://www.espressif.com/en/products/socs/esp32-s2

[6] https://github.com/espressif/esp-idf

In order to measure energy consumption and annotate the measurements via UART logs, we used the Otii Arc[7] device. Otii is a high precision power supply and analyzer unit, which comes with Otii software. Otii can be used to monitor or record real-time voltage, current, and UART logs and it is powered by USB. We supplied ESP32-S2 with 3.3V and we used baud rate 11520 for the logs.

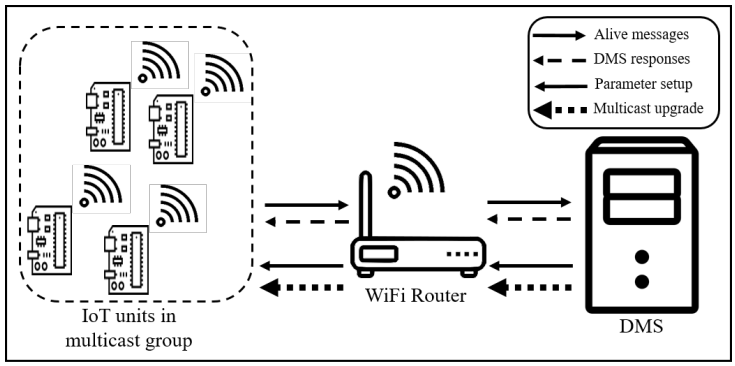


Figure 2: Information flow between DMS and IoT units

6.2 Security Choices and Package Size

In our implementation, we have used AES as an encryption algorithm and evaluated performance for key sizes of 128 and 256 bits with GCM mode and a standard IV size of 128 bits. For the hash function, we have used the SHA-256 function and for MAC, we have used HMAC with digest mode SHA-256 and key sizes of 128 and 256 bits. Hence, the hash or MAC size in our package is 32 bytes.

The new firmware size is usually less than a few Megabytes, thus, we chose 4 bytes for that which is big enough to represent the index of all chunks. In order to avoid IP fragmentation, the payload size of packages is bound to 1012 bytes, and as a result, the encrypted part of the package by excluding Hash, MAC, and index size will be 944 bytes, different field sizes are shown in Figure 1.

6.3 Testbed Setup

We have designed a testbed scenario consisting of ten ESP32-S2 boards, one DMS, and a WiFi router which is used by the IoT units to communicate with the DMS. In the test scenario, four out of ten IoT units send alive messages to the DMS and they will be added to a multicast group by DMS. In our testbed, the original firmware length is 139200 bytes and it divides into 148 chunks of 944 bytes and they were prepared by DMS for the upgrade procedure. Finally, those four IoT units will receive the multicast update and their firmware will be updated. In the

[7] https://www.qoitech.com/

test scenario, Otii is connected to one of the four IoT units and is used to record the required energy and time to complete the OTA procedure.

7 Evaluation

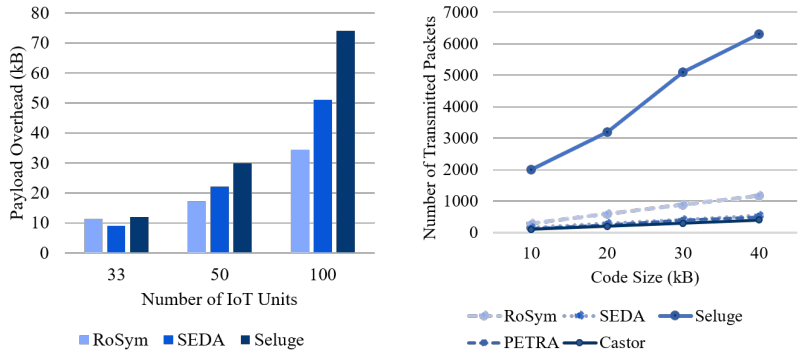
In order to show the security and efficiency of RoSym, we have evaluated it with respect to security as well as communication, computation overhead, and memory footprints.

7.1 Security

RoSym firmware upgrade protection is based on the assumption of having long term, *individual* keys shared between each IoT unit and the DMS. These can and should be updated on a regular basis, but still, if this key is compromised, the security for that *single* IoT unit is lost. As long as such attack *does not* happen, the security of the scheme will depend on the protocol for software dissemination as such. We show in Section 8.1 using ProVerif that the confidentiality and integrity of the software distribution then hold in all cases. We also show using ProVerif that under these assumptions, the DoS resistance with respect to the possibility for an attacker to get any non original firmware upgrade packet to be accepted also holds in all cases. The security of the distribution of the actual keys for the protection of the upgrades is made using standard OSCORE (see Section 5.1). Hence, the security of this part is kept as long as the OSCORE protocol and the implementation are secure.

According to our threat model, a full compromise of a single or few IoT units might happen. Under this circumstance, the confidentiality and integrity proof will not hold anymore on the packet level. However, the integrity of the final hash is protected by the *individual* IoT, long-term keys, and will not be affected. This means that compromise of a single unit, independent if it is on execution level or through key compromise by external analysis, will destroy the security of that unit only and not the software integrity of the rest of the units in the same system. In this case, DoS attacks against the complete system will be possible. However, one might argue that the effort of a direct physical attack against an IoT unit with just a DoS attack goal is less likely. Furthermore, if such happens, it will be possible to detect and it should be possible to find the compromised units in the system and replace and/or exclude them from the system. Hence, we have a reasonable trade-off between security, implementation complexity, and DoS resistance in our design.

Our design also considers the possibility for an attacker to change the delivery order of upgrade packages with the purpose of exhausting resources. Such an attack can be performed through a combination of packet modification and packet delivery delay, i.e. the attacker just modify one of the first packets, which is then delayed until the very end of the firmware load. This would then fill up the



(a) Key establishment overhead **(b)** Number of transmitted packets

Figure 3: Comparison of communication overhead in different schemes

firmware upgrade memory with almost a complete software image that is invalid. Our design has protection against such an attack by setting a maximum time value for the delay between two consecutive packets. This considerably reduces the time window for a DoS attack of this type while still allowing some out of order delivery (within the selected time threshold).

7.2 Communication Overhead

Considering communication overhead, we have compared our approach to two mixed approaches, SEDA, and Seluge, and two symmetric approaches, PETRA, and Castor in the actual update phase. We compare the key establishment phase only with SEDA and Seluge and we exclude PETRA and Castor since they do not present any key establishment method. SEDA support multicast upgrade similar to our scheme, although the key establishment phase in our scheme requires transferring the security keys to all units using an underlying secure channel. In SEDA, a group key distribution technique is instead used to share the private key, and Seluge uses pairwise key establishment with neighbor units which causes logarithmic overhead. The total number of bytes need to be transferred to each IoT unit in our key establishment phase are 108 bytes including IK_{sw}, K_{sw}, h_0, T_1, T_2, and n. We included the byte overhead of OSCORE in our key establishment phase as well. As it can be seen in Figure 3a, our key establishment phase is more efficient in medium to large sized networks in comparison to SEDA and Seluge (the figures of SEDA and Seluge schemes are taken from [Kim+16]). The efficiency of SEDA in a small sized network is due to the use of multicast group transfer.

In Seluge the payload size is bound to 102 bytes and in order to have a fair comparison of the number of transmitted packets for different code sizes in the update phase, we bound the payload size of SEDA, PETRA, Castor, and our scheme to 102 bytes, although we can send larger payloads in our scheme. In Seluge, SNACK

(Selective Negative ACK) packets, hash packets, and data packets are included in the communication overhead. Seluge uses sequential packet delivery along with a *Merkle hash tree* for integrity protection. This sequential delivery increase the number of SNACK packets which further results in an increase in communication overhead and as can be seen in Figure 3b, Seluge has the highest overhead in the number of transmitted packets. Figure 3b represents RoSym has a slightly higher communication overhead with respect to the number of transmitted packets compared to SEDA (The figures of SEDA and Seluge schemes are taken from [Kim+16]). This is due to the fact that SEDA uses SHA-128 bits and HMAC with SHA-128 bits which causes lower byte overhead in comparison to our scheme which uses SHA-256 bits for both hash function and HMAC. This allows SEDA to have more data payload in each packet and therefore the total number of transmitted packets will be reduced. Hence, at an equal security level, SEDA's overhead would be almost equal to our scheme. Castor and PETRA have the lowest byte overhead among all, this is due to the fact that in Castor instead of packet level hashes, page level hashes and MACs are used and more data payloads can be sent in each packet. PETRA also avoids MAC verification of individual packets and it uses bloom-filter along with the MAC verification of the whole update package. Although Castor and PETRA have the lowest byte overhead avoiding packet level verification makes them vulnerable to DoS attacks.

7.3 Computation Overhead

The energy consumption and required time from receiving the first upgrade package until the last have been measured using the Otii device and the results are shown in Table 1. All the measurements are the average values over 10 times upgrade using the ESP32-S2 and the Otti Arc device. The required time and energy for session setup and rebooting the IoT unit are disregarded in the measurements. As can be seen in Table 1, RoSym almost doubles both energy consumption and time compared to a completely unprotected upgrade. The UDP row in the Table is the case without any added security.

As mentioned earlier, ESP32-S2 has cryptographic hardware acceleration support both for AES encryption and hash functions including SHA-256. We have performed measurements both with and without using them. The results of the measurements are shown in Table 1. Using hardware acceleration for RoSym with the symmetric key in the case of AES 128 and AES 256, in the case of HMAC with the key size 128 bits saves 28 μwh and 27 μwh energy and reduces the required time by around 0.67 seconds and 0.31 seconds respectively in comparison to the case when hardware acceleration was disabled. Though, the comparison of RoSym in the case of AES and HMAC 128 bits to AES and HMAC 256 bits (with and without hardware acceleration) does not indicate any significant difference.

For comparison, we tried RoSym with an asymmetric key as well. We modified RoSym in the following way: we encrypted a 128 or 256 bits symmetric AES

key with RSA key size 2048 bits and transferred it to the ESP32-S2 units. The recipient units can then access the AES key after asymmetric decryption. We removed HMAC verification and added RSA signature verification instead. This setup with 128 bits AES gave 1170 μwh, 15.88 seconds with hardware accelerator, and 1200 μwh, 15.94 seconds without hardware accelerator, respectively. The figures for RSA with 256 bits AES resulted in 1170 μwh, 15.87 seconds, and 1210 μwh, 15.96 seconds, with and without hardware accelerator.

7.4 Memory Footprints

The on-chip memory on ESP32-S2 includes 320 kB SRAM and 128 kB of ROM on the MCU and it has also 4 MB SPI flash and 2 MB PSRAM. The D/IRAM is a part of RAM that can contain both data and executable data. In each round of our scheme, other than the newly received package, the hash of the next package is stored in D/IRAM memory. The flash memory is used to store the whole decrypted firmware packages, other than that, some other information including IK_{sw}, K_{sw}, the hash of the first chunk, timing information, etc. need to be stored in flash memory. The memory footprints of our scheme and MbedTLS library (used for cryptographic operations) on ESP32-S2 are shown in Table 3.

Table 2: Required energy and time for OTA using RoSym on ESP32-S2

Type	Key Size (bits)	Hardware Acceleration	Energy (μwh)	Time (s)
RoSym	128 AES, 128 HMAC	✓	848	12.19
	128 AES, 128 HMAC		876	12.86
	256 AES, 128 HMAC	✓	836	12.18
	256 AES, 128 HMAC		863	12.49
	128 AES, 256 HMAC	✓	831	12.19
	128 AES, 256 HMAC		846	12.78
	256 AES, 256 HMAC	✓	833	12.21
	256 AES, 256 HMAC		871	12.81
UDP	-	-	419	5.74

Table 3: Memory usage of RoSym on ESP32-S2

Module	Flash (kB)	D/IRAM (kB)
RoSym OTA scheme	97.7	11.1
MbedTLS library	8.4	0.05
Total	106.1	11.15

8 Formal Security Verification

We formally model and verify the security properties of our designed solution using ProVerif [Bla+18] tool. ProVerif is an automated tool that is used in verifying the security properties of protocols and it uses Dolev-Yao model [DY81] as the adversarial model. In ProVerif applied pi calculus [RS11] is used as the modeling language.

8.1 ProVerif Modeling

For modeling RoSym with ProVerif, we first declared types, variables, functions, assumptions, queries, events, and processes. We have modeled DMS and two sample devices in ProVerif as top level processes named *DMS*, *deviceA*, and *deviceB*. Thus, RoSym is encoded using a main process and three other process macros to represent *DMS*, *deviceA*, and *deviceB*. The process macros are defined as (!process) in the main process.

In the model, we define free names which are globally known, and [private] names which are not known to the attacker. We assume the DMS to be fully trusted and can not be compromised. Furthermore, we do not, in general, assume that IoT units are compromised, as a result, the attacker can not access the [private] names defined in the model including device individual key K_u or authentication and encryption keys (IK_{sw} and K_{sw}) stored on the DMS.

The main functions of the protocol including MAC, symmetric encryption, and decryption are modeled as constructors and the destructor below, both *mac* and *sencrypt* take two arguments of type key and bitstring and they return an argument of type bitstring which is either the MAC or the encryption output. The decryption function is modeled as the destructor *sdecrypt* represented below:

```
fun mac(key,bitstring) : bitstring.

fun sencrypt(bitstring,key): bitstring.
reduc forall x:bitstring, y:key;
sdecrypt(sencrypt(x,y),y) = x.
```

There are a number of events defined in the model including initiating and terminating device and the DMS server representing as: *initserver*, *initDevice*, *termDevice*, and *termserver*. The relationship between these events is denoted as correspondence assertion.

After modeling RoSym in ProVerif we have verified the following security properties using ProVerif: 1) Confidentiality of software packages, 2) Integrity of packages and 3) DoS resistance. In order to verify these security properties, different properties in ProVerif including secrecy property, correspondence assertion, and authentication property were used and all of these properties had been successfully verified in ProVerif as we show below.

8.2 ProVerif Verification

Confidentiality

We used the secrecy property in ProVerif to verify the confidentiality of the keys IK_{sw} and K_{sw} and the plain software package *I*. The secrecy property is specified using the queries below:

```
query attacker ( IKsw ).
query attacker ( Ksw ).
query attacker ( I ).
```

All three queries above have been successfully verified in ProVerif which indicates that the attacker can not gain any knowledge about I or even the keys IK_{sw} and K_{sw}.

Integrity

The integrity of software packages is preserved if the obtained package by all devices (from the DMS) is consistent. This means that for the same input and the same function, all of the devices should retrieve the same result. We prove this property using the correspondence queries as follows.

```
query a:key,b:key, q:bitstring;
event(termDevice(a,b,q))
==>event(initserver(a,b,q)).

query a:key,b:key,m:key,n:bitstring;
event(termserver(a,b,m,n))
==>event(initDevice(a,b,m,n)).
```

As it can be seen the input argument of these events are type *key* and *bitstring* and the key types are used to represent different keys such as IK_{sw}, K_{sw}, and device key K_u and *bitstring* type is used to represent the encrypted value of software packages. As indicated in the queries the input values on both sides are consistent and ProVerif had successfully verified these queries, as a result, the integrity of software packages is preserved.

DoS Resistance

According to the Denial of Service definition which is defined in [MWC12], a protocol is resistance to denial of service attacks if and only if all of the received messages in a set of received messages are authenticated, as a result for DoS resistance verification, we verified all messages are authenticated in RoSym. In RoSym, this authentication should be done in a limited time interval which this feature adds another layer of DoS protection. For formal verification, we only verified message

authentication in ProVerif since the time limit is out of the scope of modeling the protocol.

The authentication property is specified as different correspondence assertions, which indicates the relationships between events as if an event has been executed then another event has been previously executed. The queries defined in integrity property Section 8.2 can be used to prove DoS resistance as well. Those queries will be satisfied only if for each occurrence of the events *termDevice* and *termserver* there is a previous execution of event *initserver* or *initDevice*, ProVerif successfully verified these correspondence assertions as well, therefore DoS resistance is verified.

9 Conclusions

In this paper, we proposed RoSym an efficient, robust and DoS protected OTA upgrade procedure for IoT networks. RoSym can be used in a multicast manner as well as through direct download from local or remote upgrade servers. The latter is possible as each packet is protected individually during transfer or at intermediate storage. The scheme only uses symmetric cryptography, as required for resource constraint IoT devices. RoSym can handle out of order packet delivery without DoS risk as packets can be individually verified (with weak integrity). Strong integrity verification can be done as soon as the packet arrives in the correct order at the target device. The scheme is built upon that prior to the upgrade procedure, secret keys are transferred from a central distribution entity, the DMS, to IoT devices targeted to upgrade. For that, a secure session using OSCORE or similar can be used as we showed in our paper. Unlike other existing upgrade procedures, in our scheme, command packets are not required and IoT devices do not need to be awake continually. Instead, they send alive messages periodically and any information that needs to be sent to IoT devices will be sent as a piggyback on the response of alive messages. This feature and the use of symmetric cryptography along with a robust DoS protection technique, make our scheme robust, secure, and at the same time energy efficient even during idle times. As it is a purely symmetric solution, for larger key size choices, it provides full post-quantum resistance. Our security verification showed that RoSym offers the expected security level fulfilling the identified confidentiality, integrity, and DoS protection properties. Finally, our experiential results on ESP32-S2 confirmed the efficiency and robustness of RoSym.

Acknowledgements

This work was financially supported in part by the Swedish Foundation for Strategic Research, with the grant RIT17-0032, and in part by the Wallenberg AI, Autonomous Systems and Software Program (WASP).

References

[Ara+21] K. Arakadakis et al. “Firmware Over-the-Air Programming Techniques for IoT Networks - A Survey”. In: *ACM Comput. Surv.* 54.9 (Oct. 2021).

[AS+18] F. Afianti, T. Suryani, et al. “Dynamic cipher puzzle for efficient broadcast authentication in wireless sensor networks”. In: *Sensors* 18.11 (2018), p. 4021.

[Asc+12] N. Aschenbruck et al. “Selective and secure over-the-air programming for wireless sensor networks”. In: *2012 21st International Conference on Computer Communications and Networks (ICCCN)*. IEEE. 2012, pp. 1–6.

[Aso+18] N. Asokan et al. “ASSURED: Architecture for secure software update of realistic embedded devices”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018), pp. 2290–2300.

[Beu21] W. Beullens. *The Design and Cryptanalysis of Post-Quantum Digital Signature Algorithms*. KU Leuven, 2021.

[Bla+18] B. Blanchet et al. “ProVerif 2.00: automatic cryptographic protocol verifier, user manual and tutorial”. In: *Version from* (2018), pp. 05–16.

[BS13] S. Brown and C. Sreenan. “Software Updating in Wireless Sensor Networks: A Survey and Lacunae”. In: *Journal of Sensor and Actuator Networks* 2.4 (Nov. 2013), pp. 717–760.

[DHM06] J. Deng, R. Han, and S. Mishra. “Secure code distribution in dynamically programmable wireless sensor networks”. In: *Proceedings of the 5th international conference on Information processing in sensor networks*. 2006, pp. 292–300.

[DRA14] F. P. DRAFT. “202. SHA-3 Standard: Permutation-Based hash and extendable-output functions”. In: *Information Technology Laboratory, National Institute of Standards and Technology. Recovered on May* (2014).

[Dut+06] P. K. Dutta et al. “Securing the deluge network programming system”. In: *5th International Conference on Information Processing in Sensor Networks*. IEEE. 2006, pp. 326–333.

[DY81] D. Dolev and A. C. Yao. “On the Security of Public Key Protocols”. In: *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*. SFCS ’81. Washington, DC, USA: IEEE Computer Society, 1981, pp. 350–357.

[Fan+21] Y. Fan et al. “A Secure IoT Firmware Update Scheme Against SCPA and DoS Attacks”. In: *J. Comput. Sci. Technol.* 36.2 (2021), pp. 419–433.

[Fer+17] J. Ferreira et al. “Management of IoT Devices in a Physical Network”. In: *21st International Conference on Control Systems and Computer Science (CSCS)*. 2017, pp. 485–492.

[G S19] F. P. G. Selander J. Mattsson. *Object Security for Constrained RESTful Environments (OSCORE)*. `https://tools.ietf.org/html/rfc8613`. [Online; accessed 24-March-2021]. 2019.

[Gro96b] L. K. Grover. “A Fast Quantum Mechanical Algorithm for Database Search”. In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. STOC ’96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 212–219.

[GTH15] C. Gehrmann, M. Tiloca, and R. Höglund. “SMACK: Short message authentication check against battery exhaustion in the Internet of Things”. In: *12th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. IEEE. 2015, pp. 274–282.

[HC04] J. W. Hui and D. Culler. “The dynamic behavior of a data dissemination protocol for network programming at scale”. In: *Proceedings of the 2nd international conference on Embedded networked sensor systems*. 2004, pp. 81–94.

[He+11] D. He et al. “SDRP: A secure and distributed reprogramming protocol for wireless sensor networks”. In: *IEEE Transactions on Industrial Electronics* 59.11 (2011), pp. 4155–4163.

[He+12] D. He et al. “DiCode: DoS-resistant and distributed code dissemination in wireless sensor networks”. In: *IEEE Transactions on Wireless Communications* 11.5 (2012), pp. 1946–1956.

[Hyu+08] S. Hyun et al. “Seluge: Secure and dos-resistant code dissemination in wireless sensor networks”. In: *2008 International Conference on Information Processing in Sensor Networks*. IEEE. 2008, pp. 445–456.

[IKC09] W. Itani, A. Kayssi, and A. Chehab. “PETRA: a secure and energy-efficient software update protocol for severely-constrained network devices”. In: *Proceedings of the 5th ACM symposium on QoS and security for wireless and mobile networks*. 2009, pp. 37–43.

[JHC22] I. Joseph, P. B. Honnavalli, and B. R. Charanraj. "Detection of DoS Attacks on Wi-Fi Networks Using IoT Sensors". In: *Sustainable Advanced Computing*. Ed. by S. Aurelia et al. Singapore: Springer Singapore, 2022, pp. 549–558.

[KBC97a] H. Krawczyk, M. Bellare, and R. Canetti. *HMAC: Keyed-hashing for message authentication*. 1997.

[KD11] I. Krontiris and T. Dimitriou. "Scatter–secure code authentication for efficient reprogramming in wireless sensor networks". In: *International Journal of Sensor Networks* 10.1-2 (2011), pp. 14–24.

[KGN07] D. H. Kim, R. Gandhi, and P. Narasimhan. "Exploring symmetric cryptography for secure network reprogramming". In: *27th International Conference on Distributed Computing Systems Workshops (ICDCSW'07)*. IEEE. 2007, pp. 17–17.

[Kim+16] J. Y. Kim et al. "Seda: Secure over-the-air code dissemination protocol for the internet of things". In: *IEEE Transactions on Dependable and Secure Computing* 15.6 (2016), pp. 1041–1054.

[LGN06] P. E. Lanigan, R. Gandhi, and P. Narasimhan. "Sluice: Secure dissemination of code updates in sensor networks". In: *26th IEEE international conference on Distributed Computing Systems (ICDCS'06)*. IEEE. 2006, pp. 53–53.

[LKK15] J. Lee, L. Kim, and T. Kwon. "Flexicast: Energy-efficient software integrity checks to build secure industrial wireless active sensor networks". In: *IEEE Transactions on Industrial Informatics* 12.1 (2015), pp. 6–14.

[MWC12] B. Meng, W. Wang, and W. Chen. "Verification of Resistance of Denial of Service Attacks in Extended Applied Pi Calculus with ProVerif." In: *J. Comput.* 7.4 (2012), pp. 890–899.

[Per+02a] A. Perrig et al. "SPINS: Security protocols for sensor networks". In: *Wireless networks* 8.5 (2002), pp. 521–534.

[PUB12] F. PUB. "Secure hash standard (shs)". In: *Fips pub* 180.4 (2012).

[Qiu+17] T. Qiu et al. "A secure time synchronization protocol against fake timestamps for large-scale Internet of Things". In: *IEEE Internet of Things Journal* 4.6 (2017), pp. 1879–1889.

[Ron+17] E. Ronen et al. "IoT Goes Nuclear: Creating a ZigBee Chain Reaction". In: *IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 195–212.

[RS11] M. D. Ryan and B. Smyth. "Applied pi calculus". In: *Formal Models and Techniques for Analyzing Security Protocols*. Ios Press, 2011, pp. 112–142.

[San07] D. Sanchez. "Secure, accurate and precise time synchronization for wireless sensor networks". In: *Proceedings of the 3rd ACM workshop on QoS and security for wireless and mobile networks*. 2007, pp. 105–112.

[Shi+17] J. Shim et al. "A case study on vulnerability analysis and firmware modification attack for a wearable fitness tracker". In: *IT Converg. Pract* 5.4 (2017), pp. 25–33.

[SS21] Z. A. Sheikh and Y. Singh. "Lightweight De-authentication DoS Attack Detection Methodology for 802.11 Networks Using Sniffer". In: *Proceedings of Second International Conference on Computing, Communications, and Cyber-Security*. Ed. by P. K. Singh et al. Singapore: Springer Singapore, 2021, pp. 67–80.

[Tan+13] H. Tan et al. "A confidential and DoS-resistant multi-hop code dissemination protocol for wireless sensor networks". In: *Computers & security* 32 (2013), pp. 36–55.

[VRM21] L. Verderame, A. Ruggia, and A. Merlo. "PATRIOT: Anti-Repackaging for IoT Firmware". In: *CoRR* abs/2109.04337 (2021). arXiv: 2109.04337.

[Xie+15] M. Xie et al. "SecNRCC: a loss-tolerant secure network reprogramming with confidentiality consideration for wireless sensor networks". In: *Concurrency and Computation: Practice and Experience* 27.10 (2015), pp. 2668–2680.

Chuchotage: In-line Software Network Protocol Translator for (D)TLS

Abstract

The growing diversity of connected devices leads to complex network deployments, often made up of endpoints that implement incompatible network application protocols. Communication between heterogeneous network protocols was traditionally enabled by hardware translators or gateways. However, such solutions are increasingly unfit to address the security, scalability, and latency requirements of modern software-driven deployments. To address these shortcomings we propose Chuchotage, a protocol translation architecture for secure and scalable machine-to-machine communication. Chuchotage enables in-line TLS interception and confidential protocol translation for software-defined networks. Translation is done in ephemeral, flow-specific Trusted Execution Environments and scales with the number of network flows. Our evaluation of Chuchotage implementing an HTTP to CoAP translation indicates a minimal transmission and translation overhead, allowing its integration with legacy or outdated deployments.

Pegah Nikbakht Bideh and Nicolae Paladi. "Chuchotage: In-line Software Network Protocol Translator for (D)TLS ". In *the 24th International Conference on Information and Communications Security, ICICS 2022, Kent, Canterbury, UK*, pp. 589-607. Springer, Cham.

1 Introduction

Despite efforts towards standardization and interoperability, many applications use proprietary protocols and incompatible data models for information exchange [NAG19]. This is particularly acute to address in growing density of connected embedded devices or "things". Such devices are increasingly expected to communicate in a machine-to-machine (M2M) pattern. Communication among devices, or between devices and back-end systems that use incompatible protocols can be enabled through *protocol translation*. This is commonly realized either with hardware translators, virtual gateways[1], or distributed software applications [Zan+14]. Existing approaches for protocol translation are unfit to address the scalability, latency, and security requirements of current and emerging deployment topologies [DED17]. Such solutions display at least one of the following limitations.

1. in-line translation solutions do not support encrypted network traffic;
2. solutions to circumvent limitation (1) rely on deploying trusted certificates to unprotected devices on the network path and increase the attack surface;
3. cloud-based protocol translation solutions support translation over secure network communication by terminating TLS connections in a single centralized component. This increases communication latency between network endpoints and introduces a single point of failure.

Addressing the above challenges is a prerequisite to enable wide-scale device connectivity. This requires support for secure and fast in-line software network protocol translation of encrypted traffic; support for communication over several application layer protocols while maintaining latency requirements; and finally support for distributed protocol translation. Our goal is to enable secure massive M2M communication using protocol translation capable of dynamically adapting to new devices and network topologies. Our **contributions** are as follows:

- we introduce Chuchotage[2], an efficient and secure protocol translator architecture addressing scalability, latency, and security requirements of large-scale networks; Chuchotage builds on earlier work in Software Defined Networking, Trusted Execution Environments, and TLS interception;
- Chuchotage performs in-line protocol translation while supporting secure distributed network communication throughout the network fabric, avoiding translation in a logically or physically centralized back-end;
- we introduce flow-specific, on-demand translator boxes created by software switches on the network path for TLS interception and protocol translation.

[1] Communication servers including a virtual gateway to perform protocol translation.

[2] The term *chuchotage* is a form of interpreting where the linguist is near a small target audience and whispers a simultaneous interpretation of what is being said.

- we integrate secure protocol translation in OpenFlow [McK+08a] by reusing and extending its signaling. This allows to maintain backward compatibility.

Our solution relies on three principles: (i) secure TLS interception with the use of TEEs; (ii) high-performance confidential protocol translation, and (iii) fault-tolerant distributed architecture with the help of SDN networking. A TEE provides confidentiality and integrity with the use of an isolated execution environment. The loaded code and data to the TEE can be protected from various attacks. In our architecture, we use TEEs to securely decrypt, translate data, and re-encrypt it with a high level of confidentiality and integrity.

In SDN networking, network intelligence is logically centralized, thus abstracting the network infrastructure from network applications [Kre+14]. In SDN, the controller has a global view and can decide what suits best for the network. The OpenFlow protocol is usually used in SDN to link the controller and other components, e.g. switches, and routers. OpenFlow is compatible with both hardware and software switches. In Chuchotage, the software switch (Open vSwitch [Tu+21] in our implementation) makes informed decisions on application layer protocol translation to provide a high-performance and fault tolerant architecture. To the best of our knowledge, this is the first work that integrates datapath flow matching with secure protocol translation. To improve the performance, we introduce a cross-layer optimization for switch actions described in Section 4.

The rest of this paper is structured as follows: in Section 2 we introduce the relevant background and problem, followed by a review of the related work in Section 3. We describe the design of Chuchotage in Section 4. We discuss in Section 4 the design choices of the Chuchotage implementation, followed by performance and security evaluation in Section 7. We conclude in Section 9.

2 Background

We define interoperability in IoT networks as the capability of heterogeneous devices and applications to communicate and exchange data or services. Tolk et al. presented interoperability as a layered model with two main layers: *technical* and *semantic* interoperability [Tol04]. *Technical* interoperability enables compatibility of heterogeneous devices through common communication protocols and standards. *Semantic* interoperability enables heterogeneous services and applications to exchange information in a meaningful way [05].

Data or information models used by heterogeneous IoT devices are often incompatible, thus limiting semantic interoperability. Semantic protocol translators are a possible solution; they are able to convert information formats, allowing communication between heterogeneous endpoints. Such translators ingest a standardized way of representing vocabularies of processes or messages. However, despite ongoing efforts for IoT semantic translation, we are yet to see a unified secure

platform compatible with most common IoT protocols. We next briefly introduce several interoperability solutions.

Physical gateways: A traditional way of interoperability is the use of hardware gateways that act as an intermediate component between endpoint devices [NAG19]. Hardware gateways can translate protocols with different standards and specifications, they are commonly one-to-one protocol translators that do not scale (new protocols require adding new hardware); moreover, they require special hardware connectors, thus increasing both the overhead and complexity.

Protocol translators: Protocol translators replace traditional interoperability solutions, such as gateways; they are intermediate components that perform direct protocol to protocol translation. Depending on where the translation is done, protocol translators are either: a) cloud back-end translators or b) middleboxes. In the first case, the traffic is re-routed to the cloud back-end for translation. In the second case, a middlebox is a hardware component or software network function placed on the communication path between the endpoints.

We review existing protocol translators in Section 3.1. These translators either do not consider security or do not scale. Some perform the translation below the application layer, thus adding further network complexity.

For further information about common IoT protocols and different interoperability solutions at different protocol layers refer to Appendix 7. We propose the Chuchotage architecture to enable protocol interoperability on the application layer. We target the application layer as it has the highest impact on application performance [Gre20].

2.1 Threat Model

Our threat model considers two aspects - security of the network communication, and security of protocol translation. We assume the Dolev-Yao model [DY83], with an adversary capable of intercepting, and synthesizing any message, being only limited by the constraints of the cryptographic methods used. Considering protocol translation, we assume limited physical access to the platform, akin to the tasks of a legitimate third party user, and excluding physically modifying, probing, or monitoring the system. The adversary is capable of exploiting software vulnerabilities in the host operating system and software network components (network switch and network functions), reloading the switch binary, accessing the host memory, and starting arbitrary processes on the host. The attacker may modify any firmware of software component on the network platforms, including the hypervisor for virtualized set-ups. This threat model is aligned with the threat models of both process-based trusted execution environments (such as Intel SGX [Ana+13] or Keystone [She+20]) as well as virtualization-based trusted execution environments (AMD SEV-SNP [20], Intel TDX [YZ20] and IBM PEF [Hun+21]). The

Chuchotage architecture may be tuned to use other TEE implementations. Considering the growing diversity of TEE implementations [YZ20] and their various approaches to defending or preventing side-channel attacks, we exclude side-channel attacks. Likewise, we exclude attacks on control-plane components of SDN deployments (such as the SDN controller) or ancillary components (such as the Certificate Authority); these components are trusted and attacks on them can be prevented using best-practice operational security. Translator boxes are not trusted and translation cannot be done securely without a TEE.

3 Related work

3.1 Protocol Translation

An early work on protocol conversion was presented in 1988 by Lam [Lam88], proposing a formal model to achieve interoperability between processes with different protocols. Its' limitation was that it needs to be implemented as a process or as a low layer protocol in the physical layer, thus adding complexity and overhead to the network.

In [DED17], the authors proposed a protocol translator for industrial IoT protocols. They proposed the use of an intermediate format in order to translate more than three protocols rather than direct protocol-to-protocol translation. The solution satisfies interoperability features including transparency, scalability, reporting, verifiability, and QoS, however without addressing any security aspects, which Chuchotage explicitly addresses.

Muppet [Udd+18] is an edge-based multi-protocol switching architecture that can be used for IoT service automation. Muppet is a P4-based switch which can communicate with IoT devices using different protocols, where switches are managed by an SDN controller. Muppet was designed for translation between Zigbee [SM06] and Bluetooth low energy (BLE) [Gar+20] protocols or translation between BLE/Zigbee and IP protocols and is therefore complementary to Chuchotage, which works at the application layer. However, similar to [DED17], Muppet does not support protocol translation over TLS communication.

3.2 TLS Interception

HTTPS interception is implemented for purposes such as content filtering, malware detection, DDoS mitigation, load balancing, etc [CO20], and despite the relative maturity of the topic, research on TLS interception proxies gained further attention in recent years. The ME-TLS protocol [Li+20] supports TLS 1.3 and enables endpoints to introduce middleboxes into a session given the consent of both parties. Endpoints can control middlebox access permissions on traffic data, and verify the middlebox service chain. The protocol is based on monitoring handshake messages passively without modifying the handshake of TLS 1.3.

An implicit version negotiation mechanism in the ME-TLS handshake protocol enables it to interoperate with TLS endpoints seamlessly. However, ME-TLS requires deploying the Boneh–Franklin identity-based encryption (BF-IBE) [KG10] instead of the widely adopted Public Key Infrastructure (PKI) approach.

maTLS is an extension to TLS that allows middlebox visibility and auditability by enabling a client to authenticate all middleboxes through a dedicated *middlebox certificate*. The use of middlebox certificates eliminates the insecure practice of installing custom root certificates or servers sharing their private keys with third parties. Furthermore, the middlebox-aware TLS (maTLS) protocol enables auditing the security behaviors of middleboxes [Lee+19].

IA2-TLS [BKS20] is an encryption-based approach to enable in-line packet inspection. IA2-TLS is based on binding an *inspection key* to the random nonces that are generated by the endpoints during a TLS handshake. The advantage of this approach is the capacity to introspect traffic both inline and offline, at any location along the network path. This approach requires modifying the client and server TLS implementation. Similar to many other TLS interception approaches, it is not practical considering the lack of backward compatibility.

Considering the properties and backward compatibility of the ME-TLS protocol, we use it for the remainder of this paper as the reference TLS interception protocol. Other approaches to TLS interception are complementary.

4 Chuchotage Protocol Translator

4.1 Architecture

Figure 1 illustrates the Chuchotage architecture, relying on principles introduced in Section 1: (i) secure and protocol-compliant TLS interception; (ii) efficient confidential protocol translation; and (iii) fault-tolerant distributed architecture. The proposed architecture assumes that network switches are configured Ⓐ with an action to translate network flows between endpoints that use incompatible application layer network protocols Ⓑ (we use OpenvSwitch for implementation). When invoked, the action triggers the creation of a translator box Ⓒ in a trusted execution environment (Intel SGX in our implementation). The translator box is subsequently attested by a verifier network function and provisioned with credentials for TLS interception Ⓓ. The translator box is network-flow specific, translates subsequent communication between the endpoints Ⓔ and terminates once the network flow is cleared from the switch flow table, as described next.

Dynamic Translator Box Creation

We use TEEs to run *translator boxes* that decrypt the TLS traffic on the respective flow, use application protocol translators to convert it to the target protocol, and re-encrypt it before forwarding. A translator box is instantiated whenever the

`translation` action is triggered by a new network flow matching the flow table rule. Depending on the implementation, translator boxes are instantiated either as a child process of the switch daemon (in-switch) or external to the switch. In-switch translator boxes are instantiated by the ovs-vswitch daemon, while external translator boxes are instantiated by the network controller. Translator boxes are deployed in TEEs to ensure execution isolation, confidentiality, and integrity of packet data.

To instantiate a translator box, the parent process first invokes the creation of a TEE and deploys the translation logic configured for the pair of application layer protocols in the respective network flow. Next, a *verifier* network function attests the integrity and authenticity of the translator box [Cok+11]. Following a successful attestation, a trusted certificate authority network function provisions the cryptographic artifacts necessary for intercepting the TLS communication between endpoints. The exact artifacts depend on the approach for TLS interception, as described next in Section 4.1. The parent process of the translator box terminates it once the respective flow is evicted from the datapath cache.

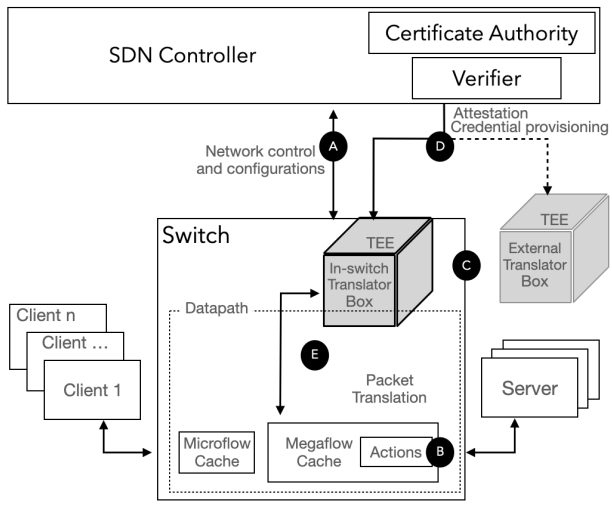


Figure 1: Conceptual illustration of the Chuchotage architecture

In our current implementation, we used Intel SGX enclaves to create TEEs. SGX enclaves rely on a trusted computing base of code and data loaded at enclave creation time. Program execution within an enclave is transparent to the underlying operating system and other mutually distrusting enclaves running on the platform. The CPU is an enclave's root of trust; it prevents access to the enclave's memory by the operating system and other enclaves. Library operating systems were used in this context to facilitate both the portability and performance of legacy applications in SGX enclaves [She+20].

TLS Interception

We focus on the TLS v1.2 [DR08b] and v1.3 [Eri18] for transport security due to their wide adoption. We further use the ME-TLS [Li+20] protocol extension for TLS interception in protocol translator boxes. The use of ME-TLS allows delivering session key materials to translator boxes in-band and does not require additional TLS connections or round-trips. Moreover, this allows retaining backward compatibility with TLS 1.3 [Eri18] through implicit protocol version negotiation. In case one of the endpoints does not support ME-TLS, communication remains encrypted but without protocol translation.

Following the TLS1.3 specification [Eri18], ME-TLS reuses the TLS 1.3 Finished message to achieve two additional goals, endpoint authentication and translator box negotiation (agreement between client and server about the translator boxes to be used). For middlebox negotiation, the `ClientFinished` and `ServerFinished` messages each contain two middlebox lists specifying the translator involved in each direction of the network path. Once both client and server endpoints complete the translator box negotiation by including the list of chosen translator boxes to the `ClientFinished` and `ServerFinished` messages, they distribute the necessary session key materials to selected translator boxes. ME-TLS achieves this through an additional `SessionKeyDistribution` message sent by the endpoints to the translator boxes on the communication path. The `SessionKeyDistribution` message is an application data message (not a handshake message); the record field of the message contains a byte sequence, which is an HMAC generated from the shared secret between the client and server (ss_{cs}^{ibe}) and a string constant to differentiate from other application data records, followed by encrypted session key materials for the translator boxes. The ME-TLS protocol uses a property of the BF-IBE scheme [KG10] that allows endpoints and translator boxes to establish a shared secret between each other through zero-round secret negotiation. In BF-IBE, a trusted authority called a private key generator (PKG) generates private keys for endpoints and translator boxes using their identities and a master key. The endpoints (client and server) can then use the shared secret to encrypt the session key materials communicated to the translator box instances.

Translator Box Integration with OvS

Translator boxes are created following the *translate* action in the flow rules and are instantiated during the transport layer protocol handshake between two communicating endpoints, regardless of the application layer protocol they use. An incoming packet to the switch is first matched against the available rules (see Appendix 7). A match against a rule that contains the *translate* action on the datapath triggers the creation of a flow-specific translator box. The translator box can be created either on the datapath in kernel space or user space, depending on the TEE implementation. When using Intel SGX, translator boxes are created in user space enclaves, since SGX enclaves can only run as user processes. While this may

affect their performance (due to IO penalties inherent to the Intel SGX model), recent work indicates that modifying software network components deployed in TEEs can help to improve their IO performance [SPV22]. Next, a verifier network function of the network controller attests the enclave to make sure it is trustworthy, then the enclave receives the key shares through key provisioning that allows it to compute session key materials and decrypt the TLS communication between the endpoints in the respective flow Figure 1. Attestation and key provisioning are done *in parallel* with the ongoing transport layer protocol handshake. All subsequent packets in the respective flow will be processed by the translator box.

Protocol to Protocol Translation

Once a translator box inside the enclave receives a packet from the respective flow, it first decrypts the packet using the session key materials computed from the key shared received from the network controller. Next, the translator box parses the decrypted packet, extracts the application data, and formats it into the destination protocol format. Finally, the formatted packet is re-encrypted and returned to the switch data path to be forwarded to its destination.

4.2 Challenges

The design of Chuchotage addresses several important challenges, namely enabling distributed protocol translation and combining TLS interception with attestation primitives of the trusted execution environments. We address distribution and scalability by introducing the concept of ephemeral, flow-specific, on-demand translator boxes created by software switches on the network path. To achieve scalability in high density networks, multiple switches, and SDN controllers can be used in the network depending on the network topology and available resources. Chuchotage combines the ME-TLS protocol for TLS interception [Li+20] with the SGX attestation protocol to provide an uninterrupted chain of trust that includes the communicating endpoints, the translator box, and the certificate authority by the communicating parties.

4.3 Operating flow

In the following operating flow description, we assume that a network administrator uses a deployment blueprint to define flow rules for the endpoints included in the topology. For the types of devices and communication protocols known beforehand, the network administrator specifies a `translate` action for the flows that require translation. Note that two distinct translation policies will be specified for each source-destination pair in a flow where endpoints implement distinct application layer protocols. In the following operating flow description, we assume the latest version of TLS, version 1.3; while other TLS versions can be made compatible with this operating flow, this requires additional adjustments.

In-line operating flow The sequence diagram in Figure 2 illustrates how translator boxes instantiated by the switch obtain the session keys negotiated between two endpoints, client and server:

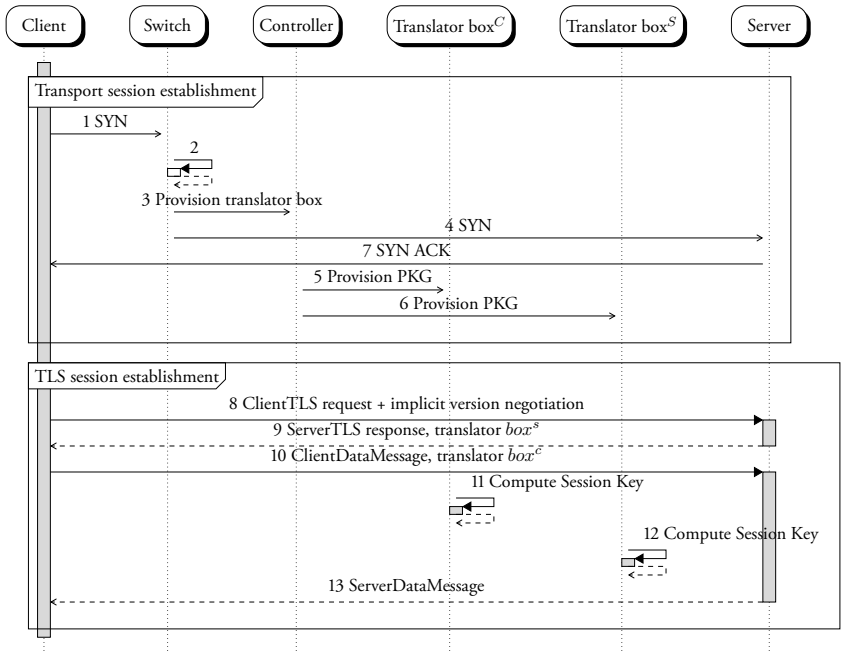


Figure 2: Chuchotage operating flow

- The Client initiates a communication session by sending a TCP `SYN` packet to Server (step 1).
- A Switch on the network path matches the `SYN` packet against entries in its Microflow cache. Since the Client did not communicate with the Server earlier, the search continues in the Megaflow cache and ultimately in the OpenFlow flow tables, where it matches the translation policy defined by the network administrator (step 2). The results of Megaflow cache lookup will be cached in Microflow cache. The switch triggers the controller to instantiate the translator boxes (step 3).
- The `SYN` packet is immediately forwarded to the destination; this avoids introducing additional latency (step 4).
- The controller instantiates translator boxes for the flows t_c (client-server, step 5) and t_s (server-client, step 6). The controller instantiates the translator box in a TEE, attests it [Cok+11; Ana+13] and provisions key shares generated by the PKG [KG10].

- The server returns a `SYN ACK` reply, the transport session is established at this point (step 7).
- The TLS negotiation starts; the negotiation follows the TLS 1.3 with the ME-TLS extensions [Li+20] (step 8). The Client TLS request includes an implicit version negotiation to check that the Server supports the ME-TLS extensions. The Server TLS response follows the TLS 1.3 specification and additionally specifies the identifier of the server translator box (step 9).
- Next the Client starts sending encrypted application data (step 10).
- The `ClientDataMessage` packet containing application data is matched in the Microflow cache of the switch and processed by translator box t_c. At this point, t_c obtains its session key material from the `SessionKeyDistribution` message and generates the key distribution bytes using the shared secret between itself and the endpoints (step 11). It derives the application traffic secrets, allowing it to derive symmetric keys to encrypt and decrypt application data on the client-server path. The session key is used for the remainder of the TLS session.
- Having decrypted the data, t_c converts the application data to Server application protocol format, re-encrypts it, and forwards the packet to the Server;
- The Server returns the application data encrypted with a TLS session key. The `ServerDataMessage` application data packet is matched in the Microflow cache of the Switch and processed by translator box t_s; t_s obtains its session key material from the `SessionKeyDistribution` message, generates the key distribution bytes using the shared secret between itself and the endpoints, and derives the application traffic secrets allowing it to derive symmetric keys to encrypt and decrypt application data on the server-client path. The session key is used for the remainder of the TLS session (step 12);
- t_s converts the decrypted application data to the client's application protocol format, re-encrypts it and forwards it to the client (step 13);
- Translation of application data continues for the remainder of the TLS session; the translator boxes are terminated once the network flow is evicted from the Switch flow cache.

In case of DTLS, the operating flow is modified such that the translator boxes are created after the `ClientHello` message.

5 Implementation

For evaluation purposes, we implemented Chuchotage with two popular IoT protocols, CoAP and HTTP. Our implementation includes the following components. A *client*, an HTTP client representing an IoT device contacts a server with a different protocol, a *Server*, A CoAP server is listening for client connections. *Open vSwitch (OvS)*: endpoints are connected to OvS through the same bridge and OvS is responsible for forwarding incoming client or server packets to the translator box, as well as forwarding outgoing packets from the translator box to their destinations; *SDN controller*: an SDN controller manages the network flows to improve network performance. For that we used Ryu[3], an open source controller. Whenever OvS does not find any matching entry in its flow caches to handle packets in need of translation it contacts the controller, which will trigger a translation. *Translator box*: via the translation process, the controller creates a translator box responsible for translating the traffic between client and server. In the translator box, we used an HTTP to CoAP parser/formatter library[4], capable of parsing and converting HTTP to CoAP messages and vice versa. *TEE*: to ensure execution isolation as well as confidentiality and integrity during packet translation, we ported the protocol translator to an SGX enclave using the Occlum library OS [She+20]. Occlum[5] is a memory-safe library OS for SGX. Note that for implementing other protocol translation (other than CoAP and HTTP), a new parser/formatter is required but the rest of the components will remain unchanged.

5.1 Implementation choices

In Chuchotage, the translator box can be instantiated either by the network controller (external) or OvS (in-switch [SPV21]). In our prototype implementation, the SDN controller deploys an SGX enclave with the translator code and attests it, as deploying, managing, and debugging external translators is easier for network administrators. Attestation can be done locally or remotely based on the location of the appraiser and of the target enclave [Cok+11]. In our prototype implementation, the SDN controller (appraiser) and translator box (target) both exist on the same platform and hence we used local attestation with a trusted enclave that exists on the SDN controller and keypair provisioning. As mentioned above, the TEE hosting the translator box can be instantiated using several alternative approaches, both virtualization-based [YZ20] or process-based [Lee+20; McK+13]. Enterprise deployments should consider remote attestation of translator boxes, or a combination of both as supported by some virtualization-based TEEs [Hun+21]. The

[3] https://ryu-sdn.org/

[4] https://github.com/keith-cullen/FreeCoAP

[5] https://github.com/occlum/occlum

choice of TEEs depends on constraints on application portability, security, performance, etc.

For TLS interception, we assume that session key materials are distributed to the involving parties including the client, server, and SDN controller prior to the handshake procedure and ME-TLS overhead is explicitly excluded in our evaluation since it only affects the handshake, not the actual communication.

Our translation policy is defined by using features extracted from the traffic flow, namely a combination of specific source and destination IP addresses and port numbers. When an incoming flow matching these features triggers the translation action and the packets in the matching flow are forwarded to the translator. After translation, the packets are sent back to the switch to be forwarded to their own destination. While distinct translator boxes can be created for inbound and outbound flows (client to server or server to client, see Section 4.3), we use one translator box for both in- and outbound flows.

5.2 Testbed

Our testbed consists of four docker containers representing client, server, a Ryu controller, and a translator box deployed in an SGX enclave (see Figure 3), as it can be seen in Figure 3 the testbed is compatible with different pairs of clients and servers. OvS was installed on the host OS and the four docker containers are connected to the OvS via one bridge (`br0` in Figure 3). Each container is connected to the bridge through its own virtual interface, indicated as `vethp` in Figure 3. Whenever a flow needs to be translated, Ryu creates and attests an SGX enclave inside container 3. The translation is done inside the enclave and the flow to be translated is afterwards forwarded through container 3.

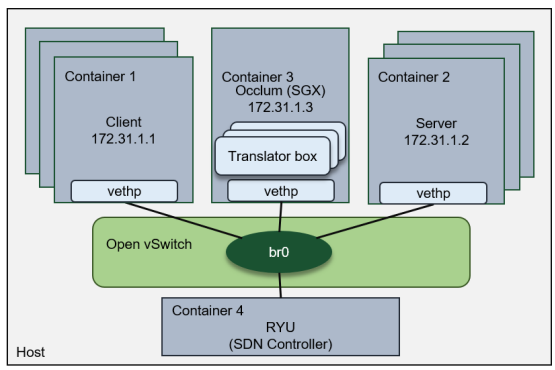


Figure 3: Testbed Overview

6 Evaluation

6.1 Performance Evaluation

We conducted several tests to evaluate the performance of Chuchotage. In the first test, we send packet batches of different sizes (100, 1000, and 10000 packets) from the client to the server and measured the translation time for the entire batch. We also measure the transmission time, i.e. the time between sending the first and last packets excluding the handshake. In this test, the client sends empty `HTTP GET` messages translated to CoAP confirmable `Reset` messages.

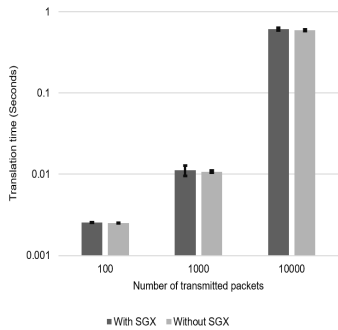
We measured translation and transmission time both with and without SGX, to measure the effect of the TEE on the performance (see Figure 4). Without a TEE, the translator box is created inside container 3 in Figure 3. As illustrated in Figure 4, both translation and transmission times slightly increase with the use of a TEE (Intel SGX in this prototype); however, this increase is acceptable in most IoT networks considering the added benefit of protecting network traffic confidentiality. Error bars are based on standard deviation.

We also compared our results with the transmission time of a vanilla CoAP to CoAP communication. Confirmable CoAP Reset messages were sent from a CoAP client to the CoAP server. The transmission time for transferring 100, 1000, and 10000 packets respectively are: 0.00719, 0.07428, and 0.70909 seconds. We consider these values as a reference point for the added overhead by the translation procedure compared to a vanilla CoAP to CoAP transfer.

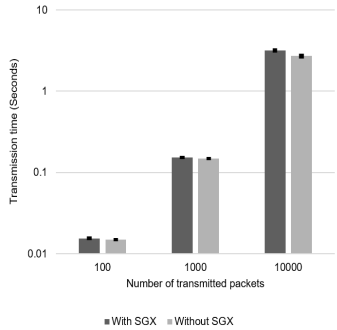
In a second test, we send batches of 100 packets of different sizes (128, 256, and 512 Bytes) from client to server and record their translation and transmission time with and without using a TEE. In this test, we send `HTTP POST` requests from the client to the server and they are translated to CoAP confirmable `POST` requests. The results of this test show that using a TEE (Intel SGX in this prototype) results in increasing both the translation and transmission time (see Figure 5). Packet data length does not affect the translation time.

In a third test, we measured the time to complete a successful handshake. The handshake takes place between the client, server, and translator box; however, the translator box is transparent for the client and server. The overall handshake time (an average of 10 handshakes) including local attestation (0.0164 seconds), enclave creation (0.80410 seconds), and additional communication between the Chuchotage components averages 2.83574 seconds. This is roughly equal to transferring and translating 10000 packets; a vanilla CoAP to CoAP handshake averages to 0.000907 seconds. However, the handshake is only performed once before translating all subsequent packets in the flow.

The performance of our proposed protocol translator is not comparable to centralized approaches, such as gateway or proxy-based approaches, since they are not suitable for large heterogeneous distribution deployments and often do not consider security of network traffic. Chuchotage is not also comparable to other existing protocol translation solutions, as earlier highlighted in Section 3.1.

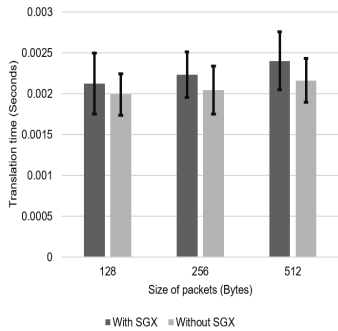


(a) Translation time

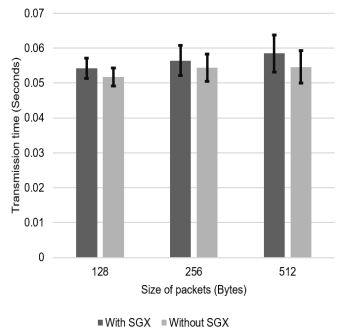


(b) Transmission time

Figure 4: Translation and Transmission Time of Translating Different Number of Packets



(a) Translation time



(b) Transmission time

Figure 5: Translation and Transmission Time of Translating Different Packet Sizes

6.2 Security Evaluation

Reflecting the structure of the threat model (Section 2.1) we discuss the security of network communication and of protocol translation.

Network Security Chuchotage uses TLS 1.3 [Eri18] to implement transport layer security - including key establishment - and inherits its confidentiality and integrity properties. On the other hand, Chuchotage also inherits any potential vulnerabilities yet to be discovered in TLS 1.3 ; this underscores the importance of following vulnerability management best practices. The security of ME-TLS extensions to TLS 1.3 is reviewed in detail in [Li+20]. There are several types of network based attacks that can target Chuchotage, such as Denial of Service (DoS) or traffic flooding. Similar to other contexts, DoS attacks can be mitigated by DoS

prevention techniques including intrusion detection and prevention systems, using load balancers, filtering, etc.

Protocol Translation Availability of a Chuchotage deployment can be ensured through network deployment best practices. High availability is an inherent capability of Chuchotage as translator boxes are instantiated and deployed in TEEs by switches throughout the network topology.

Translator boxes are central to the security of protocol translation and network communication in Chuchotage. Integrity of the protocol translation software deployed in translator boxes is verified through attestation [Cok+11]. The chain of trust evaluated through attestation is specific to the platform implementation of the TEE. During protocol translation confidentiality of provisioned cryptographic material and intercepted network traffic is ensured through TEE isolation mechanisms that include memory isolation on hardware or firmware level, run-time memory encryption, and cache flushing upon execution transition [YZ20].

In our current prototype implementation, we use Intel SGX enclaves as a TEE implementation target. SGX is vulnerable to a wide category of attacks reviewed in [NBB20]. Chuchotage can be vulnerable to any attacks applicable to SGX. However, there are a number of mitigation techniques that can be used to mitigate attacks on realistic applications deployed in SGX enclaves [Hos+18].

7 Conclusion

In this paper, we proposed Chuchotage, an in-line application layer protocol translator with transport layer security. Chuchotage relies on secure TLS interception, efficient protocol translation, and fault-tolerant distributed architecture. In Chuchotage we translate, and re-encrypt network flows with minimal latency, on the network path. Scalability is guaranteed by growing the number of translator boxes with the number of flows; translator boxes are instantiated by individual software network switches in the deployment. Depending on the capabilities of the underlying platform and their support for TEEs, Chuchotage allows creating translator boxes either in-switch or external to the switch, in kernel space or user space. We implemented a Chuchotage prototype for HTTP to CoAP translation with Intel SGX enclaves and Open vSwitch. Our evaluation indicates a slight increase in the translation and transmission time. This overhead depends primarily on the choice of TEE in the implementation.

Acknowledgments

This work was financially supported in part by the Swedish Foundation for Strategic Research, with the grant RIT17-0035, and by the Wallenberg AI, Autonomous Systems and Software Program (WASP).

References

[05] *Semantic Integration & Interoperability Using RDF and OWL*. https://www.w3.org/2001/sw/BestPractices/OEP/SemInt/. [Online; accessed 15-October-2020]. 2005.

[20] *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. White paper. Advanced Micro Devices, Jan. 2020.

[Ana+13] I. Anati et al. "Innovative technology for CPU based attestation and sealing". In: *Proc. of the 2nd international workshop on hardware and architectural support for security and privacy*. Vol. 13. ACM New York, NY, USA. 2013, p. 7.

[BKS20] J. Baek, J. Kim, and W. Susilo. "Inspecting TLS Anytime Anywhere: A New Approach to TLS Interception". In: *Proc. of the 15th ACM Asia Conference on Computer and Communications Security*. 2020, pp. 116–126.

[CO20] X. de Carné de Carnavalet and P. C. van Oorschot. "A survey and analysis of TLS interception mechanisms and motivations". In: *arXiv e-prints* (2020), arXiv–2010.

[Cok+11] G. Coker et al. "Principles of remote attestation". In: *International Journal of Information Security* 10.2 (2011), pp. 63–81.

[DED17] H. Derhamy, J. Eliasson, and J. Delsing. "IoT interoperability—on-demand and low latency transparent multiprotocol translator". In: *IEEE Internet of Things Journal* 4.5 (2017), pp. 1754–1763.

[DR08b] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246 (Proposed Standard). RFC. Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919. Fremont, CA, USA: RFC Editor, 2008.

[DY83] D. Dolev and A. Yao. "On the security of public key protocols". In: *IEEE Trans. on information theory* 29.2 (1983), pp. 198–208.

[Eri18] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. RFC. Fremont, CA, USA: RFC Editor, 2018.

[Gar+20] M. E. Garbelini et al. "SweynTooth: Unleashing Mayhem over Bluetooth Low Energy". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 2020, pp. 911–925.

[Gre20] B. Gregg. *Systems Performance, 2nd Edition*. London: Pearson, 2020.

[Hos+18] S. Hosseinzadeh et al. “Mitigating branch-shadowing attacks on Intel SGX using control flow randomization”. In: *Proc. of the 3rd Workshop on System Software for Trusted Execution*. 2018, pp. 42–47.

[Hun+21] G. D. H. Hunt et al. “Confidential Computing for OpenPOWER”. In: EuroSys ’21. Online Event, United Kingdom: ACM, 2021, pp. 294–310.

[KG10] A. Kate and I. Goldberg. “Distributed Private-Key Generators for Identity-Based Cryptography”. In: *Security and Cryptography for Networks*. Ed. by J. A. Garay and R. De Prisco. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 436–453.

[Kre+14] D. Kreutz et al. “Software-defined networking: A comprehensive survey”. In: *Proceedings of the IEEE* 103.1 (2014), pp. 14–76.

[Lam88] S. S. Lam. “Protocol conversion”. In: *IEEE Trans. on Software Engineering* 14.3 (1988), pp. 353–362.

[Lee+19] H. Lee et al. “maTLS: How to Make TLS middlebox-aware?” In: *NDSS*. 2019.

[Lee+20] D. Lee et al. “Keystone: An Open Framework for Architecting Trusted Execution Environments”. In: *Proc. of the Fifteenth European Conference on Computer Systems*. EuroSys ’20. Heraklion, Greece: ACM, 2020.

[Li+20] J. Li et al. “ME-TLS: Middlebox-Enhanced TLS for Internet-of-Things Devices”. In: *IEEE Internet of Things Journal* 7.2 (2020), pp. 1216–1229.

[McK+08a] N. McKeown et al. “OpenFlow: Enabling Innovation in Campus Networks”. In: *SIGCOMM Comput. Commun. Rev.* 38.2 (2008), pp. 69–74.

[McK+13] F. McKeen et al. “Innovative instructions and software model for isolated execution.” In: *Hasp@ isca* 10.1 (2013).

[MPA19] J. Medina, N. Paladi, and P. Arlos. “Protecting OpenFlow using Intel SGX”. In: *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2019, pp. 1–6.

[NAG19] M. Noura, M. Atiquzzaman, and M. Gaedke. “Interoperability in internet of things: Taxonomies and open challenges”. In: *Mobile networks and applications* 24.3 (2019), pp. 796–809.

[NBB20] A. Nilsson, P. N. Bideh, and J. Brorsson. “A survey of published attacks on Intel SGX”. In: *arXiv preprint arXiv:2006.13598* (2020).

[She+20] Y. Shen et al. “Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX”. In: *Proc. of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’20. Lausanne, Switzerland: ACM, 2020, pp. 955–970.

[SM06] S. Safaric and K. Malaric. “ZigBee wireless standard”. In: *Proc. ELMAR 2006*. 2006, pp. 259–262.

[SPV21] J. Svenningsson, N. Paladi, and A. Vahidi. “Faster Enclave Transitions for IO-Intensive Network Applications”. In: *Proc. of the ACM SIGCOMM 2021 Workshop on Secure Programmable Network INfrastructure*. SPIN ’21. Virtual Event, USA: ACM, 2021, pp. 1–8.

[SPV22] J. Svenningsson, N. Paladi, and A. Vahidi. “SGX-Bundler: speeding up enclave transitions for IO-intensive applications”. In: *The 22nd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*. IEEE-Institute of Electrical and Electronics Engineers Inc. 2022.

[Tol04] A. Tolk. “Composable mission spaces and M&S repositories–applicability of open standards”. In: *Spring simulation interoperability workshop, Arlington (VA)*. Citeseer. 2004.

[Tu+21] W. Tu et al. “Revisiting the Open VSwitch Dataplane Ten Years Later”. In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. SIGCOMM ’21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 245–257.

[Udd+18] M. Uddin et al. “SDN-based multi-protocol edge switching for IoT service automation”. In: *IEEE Journal on Selected Areas in Communications* 36.12 (2018), pp. 2775–2786.

[YZ20] J. Yao and V. Zimmer. “Virtual Firmware”. In: *Building Secure Firmware: Armoring the Foundation of the Platform*. Berkeley, CA: Apress, 2020, pp. 459–491.

[Zan+14] A. Zanella et al. “Internet of things for smart cities”. In: *IEEE Internet of Things journal* 1.1 (2014), pp. 22–32.

Appendix

Common IoT Communication Protocols

In the TCP/IP network model, the physical or data link layer is responsible for physical transmissions; characteristics of applications - such as latency and availability - directly impact traffic characteristics on the link layer. The network layer

is responsible for routing and forwarding packets; considering that IoT devices are often resource-constrained, the information necessary for routing should be kept at a minimum. Finally, transport layer protocols (such as TCP and UDP) manage end-to-end communication between network endpoints.

Physical network gateways are commonly used for interoperability in the physical and network layers or transport layer [NAG19]. Gateways have limited scalability [NAG19]: as the number of IoT devices increases, special connectors are required for their interaction, thus adding both cost and complexity to the network.

Application communication between network endpoints is implemented on the application layer. Middleware can perform translation in the application layer; however, connecting middleware components risks further reducing interoperability by locking applications to a specific technology. Interception proxies are an alternative for application layer translation; however, proxies cause delays since all traffic transits through proxies even when translation is unnecessary [DED17].

Proxies and middleware currently available for application layer protocol translation are increasingly unsuitable for secure, distributed, and transparent application layer protocol translation.

Several application layer protocols - namely HTTP, CoAP, MQTT, and AMQP - have been widely reviewed in academic publications and adopted in large scale deployments. We compare these protocols in Table 1.

Table 1: IoT protocols comparisons

IoT protocols	HTTP	CoAP	MQTT	AMQP
Transport layer	TCP	UDP	TCP	TCP
Security	TLS/SSL	DTLS	TLS/SSL	TLS/SSL
Architecture	Req/Res	Req/Res	Pub/Sub	Pub/Sub
QoS	No	Yes	Yes	Yes
Low Power/Lossy Networks	Fair	Excellent	Fair	Fair
Dynamic discovery	No	Yes	No	No

Open vSwitch Overview

OpenvSwitch (OvS) is an open source programmable switch [Tu+21] that implements packet forwarding on the datapath; it is a flow-based switch, where clients install flows determining forwarding decisions. Flows are installed in a cache level structure that assists the datapath to execute actions on received packets, e.g. allow, drop, etc. For each ingress packet, the datapath consults its cache and forwards the packet to its destination if matching entries exist. For each cache miss, the datapath issues an upcall and forwards the packet to ovs-vswitchd. A datapath can be deployed as a kernel module or in user space with additional firmware support. Packet classification in OvS is computationally expensive, mostly due to the

many types of matching fields. Matching is implemented in a hash table of flow rules, with matching fields hashed as keys. OvS uses a modified Tuple Space Search (TSS) algorithm for packet classification. The algorithm searches through the hash map tables based on the maximum entry's priority and terminates after finding the highest priority matching flow rule. Early OvS releases implemented OpenFlow processing exclusively as a kernel module. However, the difficulty of developing and updating kernel modules motivated moving packet classification to user space. A multi-level cache structure kernel implementation compensates the resulting performance impact. The cache structure consists of two levels with increasing lookup costs: a microflow cache (or Exact Match Cache) and a larger megaflow cache. The megaflow cache matches multiple flows with wildcards [MPA19].

Open vSwitch Forwarding Figure 6 illustrates the OvS internals. An incoming packet reaches the datapath from either a physical or virtual NIC (1). In the datapath, the switch runs a first search based on an exact match (2). If there is a matching entry in the microflow cache, the packet is sent to the specific table in the megaflow cache to retrieve the required actions. Otherwise, the forwarding process performs a second search in the next cache line (3). Failing to find a match, the datapath uses upcalls (4) to inform the ovs-vswitchd that it cannot handle the packet. The ovs-vswitchd uses the classification process (5) to obtain a matching rule via its flow tables. Next, ovs-vswitchd returns to the datapath, inserts the entry in the cache (6), and returns the packet to the kernel (7). Finally, the datapath forwards the packet to the intended destination (8). Failing to find matching information in the flow tables, ovs-vswitchd sends a packet-in request to the network controller to get a matching rule for the unknown packet.

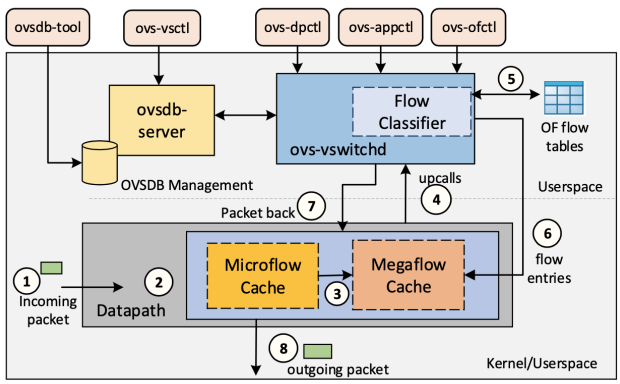


Figure 6: An overview of Open vSwitch internals

LMGROUP: A Lightweight Multicast Group Key Management for IoT Networks

Abstract

Due to limitations of IoT networks, including limited bandwidth, memory, battery, etc., secure multicast group communication has gained more attention, and to enable that, a group key establishment scheme is required to share the secret key among the group members. The current group key establishment protocols were mostly designed for Wireless Sensor Networks, requiring device interaction, high computation costs, or high storage on the device side. To address these drawbacks, in this paper we design LMGROUP, a lightweight and multicast group key establishment protocol for IoT networks, that is based on Elliptic Curve Integrated Encryption Scheme and HMAC verification and does not require device interaction. We also suggest an algorithm for unpredictable group member selection. Our experimental result of implementing LMGROUP indicates it has low storage, low computation, and low communication costs. Furthermore, the formal security verification indicates LMGROUP is secure and robust against different attacks.

Pegah Nikbakht Bideh. "LMGROUP: A Lightweight Multicast Group Key Management for IoT Networks". In *the 17th International Conference on Information Security Practice and Experience (ISPEC 2022), Taipei, Taiwan*.

1 Introduction

IoT networks have several challenges; one of the challenges is that the majority of devices are resource constrained, which means that they have limited memory, battery, power, and limited computational resources. In IoT networks, bandwidth is another challenge [Sam16; CK16] since increasing the number of devices also makes the bandwidth and communication resources limited. Due to these limitations, multicast group communication has become more favorable in IoT networks since sending multicast messages to a group of devices is more efficient than sending unicast messages and overloading the network with multiple messages. Multicast group communication is particularly important, where software updates or patches are required to be sent to a group of devices simultaneously.

In order to enable multicast secure group communication, a group key needs to be established in advance. Various group key establishment methods have been proposed, which will be described in detail in Section 2. Most of these schemes have been designed for WSN (Wireless Sensor Network), which do not take the characteristics of IoT environments into account. The designed schemes for WSN require interaction between group members to get access to the shared key. In such solutions, many nodes participate in the computations, and this results in unnecessary intensive cryptographic operations, and additional packet forwarding overhead [Kim+16]. Device interaction is hard to achieve in IoT networks where devices have the least communication. Examples of such networks in smart cities or smart homes are where different sensors are placed in different places to gather information about temperature, air pollution, etc. These sensors gather data and send aggregated data periodically (in a unicast way) to a central server for further analysis, and device-to-device communication is less likely.

The central server then should be able to send control commands or updates/upgrades and patches to the sensors in a multicast way. These commands and updates should not be broadcasted to all IoT sensors for availability reasons since if any unexpected error happens, it can affect the availability of the whole network. As a result, the central server needs to group the IoT devices and decides on group membership; this can be done manually by the administrator or automatically to make the group membership less predictable to attackers. In this paper, we first suggest an automatic algorithm for unpredictable group member selection. After the selection of group members, a group key needs to be established to the group members. These group keys need to be renewed frequently due to changes in the group membership to provide forward secrecy. For that, we then design LMGROUP, a new lightweight multicast group key management scheme that can work efficiently in small to large networks. LMGROUP does not have the problems of other group key establishment schemes for WSN, including interactions between group members or heavy procedures on constrained devices. We implement LMGROUP, and the experiments indicate it has efficient memory usage, communication, and computation costs. The experiments also show the

scalability of LMGROUP. Finally, the formal security verification indicates our multicast scheme is secure against different attacks, such as replay attacks. Our main contributions are:

- We suggest an algorithm for unpredictable group member selection.
- We design a new lightweight and multicast group key management scheme based on hybrid cryptography.
- We implement LMGROUP and indicate it is scalable and has efficient memory usage, communication, and computation costs.
- We formally verify LMGROUP and indicate it is secure against different attacks.

The rest of this paper is organized as follows: in Section 2, the related work on group key establishment methods for WSN and IoT networks is presented. In Section 4, the details of LMGROUP, including the suggested group member selection algorithm and our designed scheme, are described. Implementation details are presented in Section 4. Performance evaluation and formal security verification are described in Sections 7 and 6. Finally the paper is concluded in Section 9.

2 Related Work

Various Group Key Management (GKM) methods had been proposed for WSNs and IoT networks and were extensively reviewed in [Che16; Dam+20; PPW20], in the case of the used cryptography method, they are divided into three categories: symmetric, asymmetric, and hybrid. In the case of key establishment authority, these methods can be divided into centralized and distributed methods [Che16]. Centralized methods are mostly applicable to networks with static topology, while distributed approaches are more suitable for dynamic networks where nodes have high levels of mobility and can join and leave the network quite often. Our work focuses on centralized schemes, and we do not review distributed schemes here. Among the reviewed schemes in [Che16], the most lightweight centralized approaches applicable to static small to large networks are: LKH [WGL00], S2RP [DS06], TKH [SLS10], and LEAP [ZSJ06].

LKH (Logical Key Hierarchy) is a multicast rekeying approach for WSNs. In LKH the nodes are divided into subgroups based on a logical hierarchy and a symmetric key is assigned to each leaf node. LKH has a reasonable communication cost in most WSN networks since only the existing members in the subgroup receive the rekeying messages. However, each member of the group needs to maintain the keys from the leaf to the root node path, which causes additional storage and computation cost on the node's side [WGL00]. S2RP (Secure and Scalable

Rekeying Protocol) is similar to LKH with almost the same performance results, but instead, it has added security to authenticate the rekeying messages through the use of a one-way hash function [DS06].

TKH (Topological Key Hierarchy) is another variant of LKH in which the logical key tree is mapped to the physical topology of the nodes in the network (key tree); this further reduces the communication cost of total rekeying messages. In TKH, based on the routing tree, the key tree can be constructed; the nodes attach to a parent node until they reach the group controller (sink node). Although TKH reduces the storage overhead on the node side, it does not provide any key authentication mechanism.

In LEAP (Localized Encryption and Authentication Protocol), four types of keys are established to the nodes, including an individual key, a pairwise key, a cluster key, and a global key. This scheme has low computation, communication, and storage overhead, but for broadcast authentication, it relies on μTESLA [Per+02b] which requires synchronization between nodes, but the node synchronization is heavy, and it is hard to achieve in IoT or WSN networks.

Another lightweight and decentralized group key establishment for IoT was proposed in [Dam+20]. This scheme is also based on a logical hierarchy with one Key Distribution Center (KDC) and several Sub Key Distribution Centers (SKDCs) that can be used to avoid the single point of failure problem in centralized-based schemes discussed above. Same as LKH based schemes, each device needs to store the keys from the leaf to the parent path. Again, this scheme can have high storage costs on the device side, which depend on the subgroup size.

As mentioned above, the problems of Key Tree Hierarchy-based key establishment methods are high storage, high computation costs, and inability to provide key authentication mechanisms or requirement of time synchronization, which makes them non-practical to use for IoT networks. Most of these methods depend on the contribution of members of the same group, which is difficult to handle in IoT networks, especially in networks where IoT nodes do not interact with each other.

Other than key tree-based methods, there are a variety of non-interactive key agreement protocols [Fer+18; HL10; Lee+11; Por+15; San+18] applicable to WSNs or IoT networks. For IoT networks, a secure group key establishment based on Elliptic Curve Cryptographic (ECC) operations was proposed in [Fer+18]. This work has high computation cost since the key establishment requires one signing and one signature verification on the IoT unit side.

In [Por+15], two lightweight protocols based on ECC operations were proposed, which was an improvement of the schemes proposed in [HL10; Lee+11]. The protocols provide authenticity, confidentiality, and integrity. However, they are vulnerable to replay attacks [San+18], and they have high computational costs (especially protocol 1, which requires two signature verifications on the IoT unit side), which make them non-applicable to IoT environments. As an improvement of [Por+15], the authors in [San+18] proposed a new key establishment protocol

based on the Identity-Based Credentials (IBC) mechanism and ECC operations, which is resistant to replay attacks. In this scheme, they have used HMAC verification instead of signature verification which is more applicable to IoT devices than heavy signature-based operations. Although this scheme has a lower computation cost in comparison to protocols in [Por+15], it has higher communication costs due to the increased number of transferred messages. Also, it does not consider multicast communication and group key sharing.

Considering the problems of Key Tree Hierarchy-based methods, we address these problems and suggested a scheme, LMGROUP, that does not require time synchronization, has low storage (it does not require storing all the keys from the leaves to the root) and has low communication and computation overhead. LMGROUP is a multicast authenticated key establishment mechanism with hybrid cryptography. In LMGROUP we consider the advantages of the protocol in [San+18] including HMAC verification and replay protection; we modified the second protocol presented in [Por+15], protocol 2, and proposed a new scheme that applies to IoT networks. LMGROUP will be explained in detail in Section 4.

3 Scenario and Scheme

In this section, first, we provide the assumptions about the IoT network and the use case in which the proposed scheme is most suitable, and then we present the details of LMGROUP scheme.

3.1 Assumptions

In our network, we assume two types of nodes: resource-rich nodes and constrained IoT nodes. Resource-rich nodes do not have limited storage and processing capabilities, and they can be used to perform heavy operations and are used to manage different groups of IoT nodes. Resource-rich nodes are referred to as servers throughout this paper. We assume to have a fault-tolerant centralized architecture with redundant servers available in the network. To avoid a single point of failure having redundant servers is required. We assume a network scenario in which IoT devices are stable or have low mobility. Devices can join or leave the network for any reason, e.g., physical maintenance operations, adding new devices, or removing old ones from the network. Examples of such use case scenarios are in smart buildings with smart lights or smart doors where the devices have fixed positions.

In our use case networks, device interaction is not possible. We consider that the devices will not contribute in any way to the group key establishment. In these network scenarios, since there are usually many devices available, to keep the bandwidth as low as possible, the group key establishment is preferred to be done in a multicasted way. If during multicast group key establishment, one of

the group members fails to receive or update the group key in its next contact with the server, it can retrieve the group key in a unicast way.

We assume that each device has owned a symmetric master key denoted by k_m, this key is provisioned by the network administrator in the setup phase, and the k_m keys are also stored with each device identity securely on the server. Later, in the bootstrapping step, k_m is used to extract a session key, k_s, based on a key derivation function to establish a secure session with the server. We assume that the server decides on the group members, and the devices themselves can not decide which group they want to belong to. The devices can belong to multiple groups at the same time, but messages encrypted with one group key can not be decrypted with another key.

3.2 Network scenario

In our network scenario, after the bootstrapping phase, the devices send their aggregated data encrypted with k_s keys back to the server periodically based on defined time intervals. Then after receiving an acknowledgment from the server, the devices go to sleep mode to reduce the energy consumption. The server needs to decide which devices should to be grouped; our suggested algorithm to decide on the group members will be described below in Section 3.3. After deciding on the group members, the server will piggyback a hint along with the acknowledgment of the previous message to the IoT device. This hint carries information about the new group *ID*, and it can be used as the seed of the key derivation function on the IoT unit side to extract an authentication key which is further going to be used to authenticate the group key establishment messages. The required communication between IoT devices and the server to derive the group authentication key is depicted in Figure 1. After receiving the new group *ID* by the IoT devices inside the group, they use it to derive the key K^*, which is the authentication key. The authentication key will be used during the group key establishment phase to protect the authenticity of multicast messages.

Other than the new group *ID*, the wake-up time should also be sent to the IoT devices so they can wake up at the defined time to receive the multicast group key establishment messages. To avoid heavy time synchronization procedure on IoT unit sides, instead of sending the wake-up time, the server sends two other parameters to the devices: T and a window frame. T defines the number of seconds that the device needs to wake up after receiving the acknowledgment, and the window frame (which depends on the network latency, delay, etc., will be defined by the network administrator), defines the window frame in which the device should be active. As an example, if T is 6000 and the window frame is 60, then the device needs to be active from 5940 until 6060 seconds after receiving the acknowledgment.

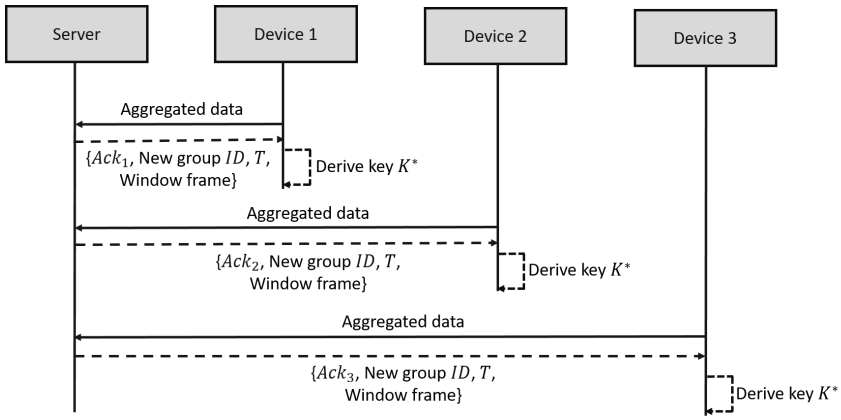


Figure 1: Communication between the server and three IoT devices in the same group to derive authentication key

3.3 Group Member Selection

In WSN or IoT networks with more mobility, the nodes themselves can join or leave the groups based on different factors, e.g., signal strength, distance, etc. There have been some methods suggested for group member selection [Kaz+11] or cluster head selection [Beh+19; HD16] in WSN. These algorithms do not apply to our use case scenario or, in general, to the networks where nodes are stable and cannot choose the grouping themselves; thus, we suggest an algorithm for group member selection that the server in our scheme can use.

The network administrator registers available IoT devices in the network to the central server (and redundant servers). On the device registration, information, including device identity number, device master secret, device public key, and device criticality level, will be stored on the server. The network administrator will decide the device's criticality level, depending on how critical the device's role is in the network. For example, in the case of smart doors in a hospital, the main entrance door has the highest criticality while the sub-doors have lower criticalities. We have considered three levels of criticality: low, medium, and high. These levels will be used for group member selection.

The group member selection should not be predictable by an outsider attacker; otherwise, the devices targeted for multicast group keying or update can become a target of DoS (Denial Of Service) attacks. Our suggested group member selection algorithm selects devices based on criticality levels and makes sure not all critical devices are in the same group. The algorithm works as follows:

As it can be seen in Algorithm 1, based on the number of available devices with different criticality levels, the members of a group will be formed randomly. If new devices join the network, they will get group membership in the next round of the algorithm. If a device leaves the network due to maintenance or replacement, the

Algorithm 1 Group member selection algorithm

Require: n, the number of group members, N, the total number of registered devices
H, M, and $L \geq 0$ are the number of devices with high to low criticality, respectively, such that $H + M + L = N$.
$G_c \leftarrow \frac{N}{n}$,
$H' \leftarrow \lfloor \frac{H}{G_c} \rfloor$, $M' \leftarrow \lfloor \frac{M}{G_c} \rfloor$, $L' \leftarrow \lfloor \frac{L}{G_c} \rfloor$, (The remaining members will be added to the groups later by the administrator.)
$i \leftarrow 0$,
while $i < G_c$ **do**
$G[i] \leftarrow$ Take random members from H, M, L with the size of H', M', and L', respectively.
$i \leftarrow i + 1$
end while
Return the groups, $G[i]$s.

group will continue with previous members until the next round of running the algorithm. Leaving a group to join another group is not possible by the device since the server only does the grouping process. The network administrator decides how often the group member selection algorithm should happen.

3.4 Designed scheme

As mentioned earlier in Section 2, two lightweight key establishment protocols based on ECC operations were proposed in [Por+15], and an improved version of them was proposed in [San+18]. The second protocol presented in [Por+15], protocol 2, and the improved protocol presented in [San+18] are the basis of our scheme. In order to better understand our designed scheme, here, we first briefly explain these two protocols and then we suggest our scheme.

Basis of LMGROUP

Protocol 2 [Por+15] uses ECIES or Elliptic Curve Integrated Encryption Scheme algorithm to establish a shared secret among the group members. In this scheme, an initiator (I) with several responders $U_j s$ in the network is considered, and the initiator determines the group members. The random r is generated by I, then R is calculated as $R = rG$ (G is the base point as in ECDH). Then for each group member EC points $S_j s$ are computed by the initiator, $S_j = d_i Q_j + R$, and Q_j represents the public key of the group members. The point $S_j = (x_i, y_j)$ will be encoded to another point (u_i, v_i) by calculating the hash over the point values. Then the encoded points will be XORed, and the results will be concatenated to make the set P. The group's secret key, k, is then the hash value over the XORed

values of u_i. The *Auth* value is calculated as *Auth = h(k || R || P)*. The multicast message that will be sent to the group members includes: *Auth, C, R, U, P*, in which C is the counter value and U contains identities of all group members. Finally, a digital signature will be added to the message, and the message will be broadcasted to all sensor nodes in the network. Each receiver first checks if its identity is included in the U part of the message; if yes, it verifies the signature and the counter value. If the verification is successful it computes u_j using *R* and its private key as: $S_j = d_j Q_i + R$. The node will encode the point, and finally, the values of the encoded point will be used to derive the group key. After that, the node verifies the authenticity of the key by checking if *Auth* is equal to $h(k \| R \| P)$. Finally, the recipient nodes will send an acknowledgment to the initiator to finish the handshake.

The problems of the above scheme are listed below:

- It is not protected against replay attacks;
- It requires heavy signature verification on the IoT unit side;
- The first message needs to be broadcasted to all sensor nodes in the network; this can cause many extra checking by IoT devices not belonging to the same group and can further cause extra overhead to the whole network.

In [San+18], the authors proposed a key management scheme that is quite similar to the work explained above [Por+15]. In [San+18] HMAC verification is used instead of signature verification which makes it more energy efficient in comparison to [Por+15]. In [San+18], replay attack protection is considered, but the scheme only works in a unicasted way, and it can not be used for group key management. The other problem of the scheme [San+18] is that, although it has lower computation overhead than the work presented in [Por+15], it causes higher communication overhead.

LMGROUP

In our designed scheme, we take the advantages of these two protocols [Por+15; San+18] including multicast and ECIES based operations for group key sharing from protocol [Por+15] and HMAC verification instead of heavy signature verification for authenticity from protocol [San+18]. We design LMGROUP that is lightweight in case of communication, computation, and storage overhead. We also suggest a replay protection mechanism and a group member selection technique so that the messages are not required to be sent to all devices in the network and can be sent to the target devices in the beginning. We describe the details of LMGROUP here.

The operation flow of our designed scheme is depicted in Figure 2. After deciding about the group members and pre-establishment of the authentication

key K^* to group members by the server, as can be seen in Figure 2, the operation flow of LMGROUP, which starts on the server side, is as follows:

- The server selects a random *r* and computes $R = rG$, in which G is the base point as in ECDH, and based on the number of group members in the range of *1 to n*, the point $S_j = dQ_j + R$ = (x_i,y_j) will be calculated, where *d* is the private key of the server and Q_j is the public key of group members.
- A unique random session *ID* will then be generated by the server that is used to protect against replay attacks.
- For each member of the group, $\overline{x_j} = \{\oplus_{i \neq j} x_i\} \oplus y_j$ will be calculated and then the set *Se* = $(\overline{x_1}, ..., \overline{x_n})$ will be formed, and the group key is calculated as $k = h(\oplus_i x_i)$ which is the hash over XOR values of x_i.
- The authentication value is calculated as *Auth = h(k* $\|$ R $\|$ $Se)$.
- The HMAC will be calculated over the fields of {*ID*, $Auth, R, Se$} with the use of the authentication key K^* and it will be sent along with {*ID*, $Auth, R, Se$} to all group members.

Then the flow continues on the device side:

- The group members upon receiving the message, first verify the HMAC, then each recipient device uses R and its own private key d_j to construct the point $S_j = d_j Q + R = (x_j, y_j)$.
- Then, the device extracts its own $\overline{x_j}$ from the received set Se, and then it can derive the group key as $k = h(\overline{x_j} \oplus x_j \oplus y_j)$, $\overline{x_j}$ contains other x and y_j value, therefore the similar values of y_j will be removed from the calculation and only all x values will be XORed as expected.
- After key derivation, the $Auth$ should be checked if it is equal to $h(k \| R \| Se)$ or not. If $Auth$ is valid then an acknowledgment will be calculated as $Ack_j = h(k \| ID \| Q_j)$, in which k is the extracted group key, *ID* is the received session *ID* by the device and Q_j is the device public key.
- The acknowledgment along with device *ID* (device identity number) will be sent back to the server.

The flow is then finalized on the server side:

- The server uses the group key and device public key Q_j to verify the acknowledgment. On a successful verification, the authenticity of the derived group key by the device is verified.

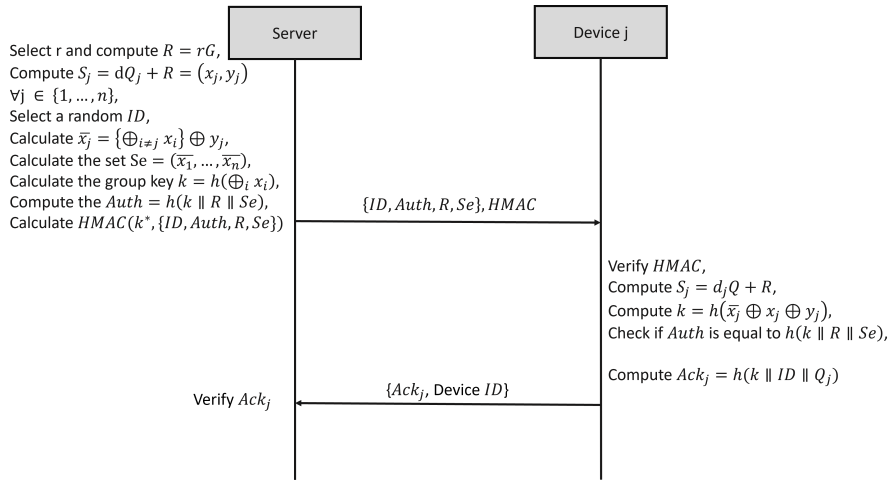


Figure 2: Operation flow between server and a sample device in LMGROUP

4 Implementation

We have implemented LMGROUP on a real testbed setup. For the implementation, we have used ESP32-S2[1] a popular IoT development board representing IoT devices (the implemented code for device side and server side is available[2]). ESP32-S2 is a low power and single-core Wi-Fi Microcontroller SoC with high performance and a rich set of IO capabilities. ESP32-S2 has cryptographic hardware accelerators for enhanced performance, and it integrates a rich set of peripherals with different programmable GPIOs that can be configured to provide USB OTG, LCD interface, UART, and other common functionalities. In our implementation, to annotate measurements, we have used UART interface. In our measurements, we used Otii Arc[3] device as a power analyzer to record and measure real-time currents and voltages using UART logs.

We have used SHA256 for the hash function, and for HMAC, we have used HMAC-SHA256. For the hash function, we have used hardware acceleration on ESP32-S2. On the IoT device side, for the ECC point addition and multiplication, we have used the Mbed TLS library.

4.1 Testbed and Environmental Setup

Our testbed consists of 10 ESP32-S2 boards, and 7 out of these 10 boards are grouped by the server to receive the group key. After the server decides the group members, it calculates: S_j, $\overline{x_j}$ (j in range of 1 to 7), Se, k. $Auth$ and HMAC

[1] https://www.espressif.com/en/products/socs/esp32-s2

[2] https://github.com/pegahnikbakht/Multi-key-share

[3] https://www.qoitech.com/otii/

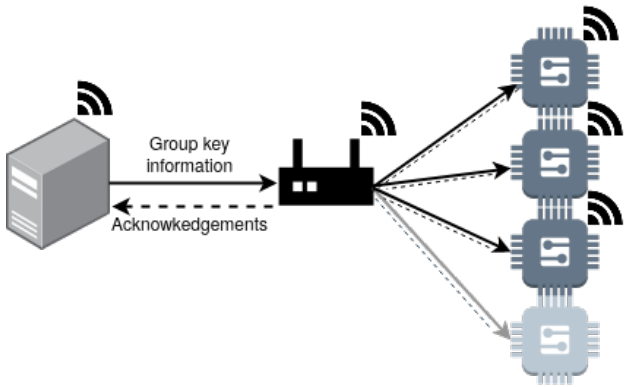


Figure 3: Testbed setup for LMGROUP

(as explained in Section 3.4). The calculation of these values can be done at any time between the time the group members have been decided until the time the devices wake up, based on the server workload during this time. During the specified wake-up time window, the devices wake up and wait to receive the group key information from the server. Our testbed setup and further communications between the server and IoT devices are depicted in Figure 4. As shown in Figure 4, the multicast group key message will be sent to all of the grouped devices by the server. After receiving, verifying the message, and extracting the group key, the devices will send back an acknowledgment to the server. Whenever the server receives the acknowledgment from all members, the group key is established and can be used for further communications. The server has a specified timeout for receiving the acknowledgments from all group members; if the server does not receive the acknowledgment from any of the group members, it will send the group key information again to those members in a unicast way later.

5 Performance Evaluation

In order to show the efficiency of our group key establishment scheme, we measure the communication, computation, and storage overhead of our scheme.

5.1 Communication and Computation Cost

In order to calculate the communication overhead, the number of transferred bytes between the server and IoT devices during the group key establishment has been calculated; in the calculations, the number of bytes in the acknowledgment is also included. The number of bytes in the first message from server to the devices are $145 + 32 * n$ (n is the number of devices inside the group), which consists of 16 bytes of *ID*, 32 bytes of HMAC, 32 bytes of *Auth*, 64 bytes of *R*, and $32*n$ bytes of

Table 1: Computation overhead of LMGROUP with two different curves

Curve		Energy (μwh)	Time (ms)
Secp256k1	Total key establishment (IoT side)	32.0809	405.7857
	Total key establishment (Server side)	-	570.3442
	Single Ack verification (Server side)	-	0.1129
	Time between arriving Acks (Server side)	-	0.8210
	Singel EC point addition and multiplication	25.6172	321.3448
Secp256r1	Total key establishment (IoT side)	39.0281	494.0000
	Total key establishment (Server side)	-	692.0681
	Single Ack verification (Server side)	-	0.1226
	Time between arriving Acks (Server side)	-	1.3483
	Singel EC point addition and multiplication	33.5333	414.1000

Se which depends to the number of group members n. The response back from the device includes an acknowledgment (32 bytes) and a *Device ID* (12 bytes) which is in total 44 bytes. Therefore the total number of transmitted bytes between the server and the devices are $189 + 32 * n$.

In order to compute the computation overhead on the device side, the energy consumption and the time was measured from the time the device receives the group key information from the server until it sends back the acknowledgment. We have done the measurements using two different elliptic curves with the same security level, Secp256r1 (prime field curve) and Secp256k1 (Koblitz curve). The total time elapsed for the key establishment on the server was also measured, which is the time from when the key establishment message was sent until all acknowledgments from the devices in the multicast group have been received and verified. The results of computation overhead are indicated in Table 1. As can be seen, a single EC point addition and multiplication of Secp256r1 consumes more energy and requires more time than Secp256k1 since prime field curves are a few bits stronger than Koblitz curves [Bjo09]. Therefore, using the prime field curve Secp256r1 for the group key establishment would require more time and energy on both the device and server sides. The required time for single acknowledge verification does not depend on the curve type, and it is almost equal for both curves, as can be seen in Table 1. Since Secp256r1 requires more processing time on the device side, it increases the arrival time of different acknowledgments as well as the total key establishment time. From Table 1, we can also conclude that the greatest part of the used energy and time on the IoT side is due to the ECC operations; therefore the hash and MAC function operations are considered negligible.

According to the measured values in Table 1, we tried to calculate how long the whole key establishment time (from the time the server sends the key establishment message until it receives and verifies all the acknowledgments from the group members) takes on the server side for larger group sizes. The results for different group sizes are depicted in Figure 4. As can be seen, the key establishment time increases by increasing the group size, and for large group sizes (larger than

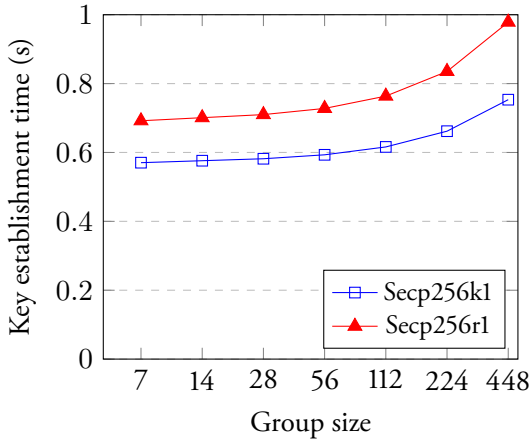


Figure 4: Total key establishment time of LMGROUP for different group sizes

100), this increase is more noticeable. Although the key establishment time for larger group sizes increases but for instance, this increase for a group of 448 nodes is still less than a second; hence LMGROUP is scalable to large-size networks as well. Note that based on the latency threshold in the network [RPC18], appropriate group size should be selected. As mentioned earlier curve Secp256k1 has better performance than Secp256r1 as can be seen in Figure 4.

We have also compared the computation and communication overhead of LMGROUP to the schemes presented in [Por+15; San+18], and the results are depicted in Table 2. Different operations are indicated as follows: *PM* for ECC point multiplications, *PA* for point addition, *h* for hash function operation, *SV* for signature verification, *HM* for HMAC operation, *MM* for modular multiplication, *SE* for symmetric encryption, and *SD* for symmetric decryption. As the hash function and HMAC operations are negligible, we can conclude from Table 2 that the protocols presented in [Por+15; San+18] obviously have more computation overhead than LMGROUP due to heavy signature verification operation and also more ECC or modular operations in schemes [Por+15; San+18], respectively. In scheme, [Por+15], the authors have originally used the curve Secp160r1, which has less byte overhead than Secp256r1 or Secp256k1 (the curves used in LMGROUP), for the scheme [Por+15] to be comparable with our scheme, we have also considered a curve with 256 bit modulus in [Por+15]. The results of the communication overhead comparison are also indicated in Table 2.

Considering the communication overhead represented in Table 2, we calculate the communication overhead for different values of n or the group size, and the results are depicted in Figure 5. As can be seen, LMGROUP has slightly higher overhead than the protocol [Por+15] and this is due to the fact that in scheme [Por+15] SHA128 was used as the hash function which generates 16 bytes lower

Table 2: Communication and computation overhead comparison of LMGROUP with the protocols presented in [Por+15; San+18]

	LMGROUP	Protocol [Por+15]	Protocol [San+18]
Computation Overhead (number of operations)	PM+PA+3h+HM	PM+PA+5h+SV	2PM+2MM+ h+2HM+SE+SD
Communication Overhead (number of bytes)	$189 + 32 * n$	$146 + 18 * n$	$128 * n$

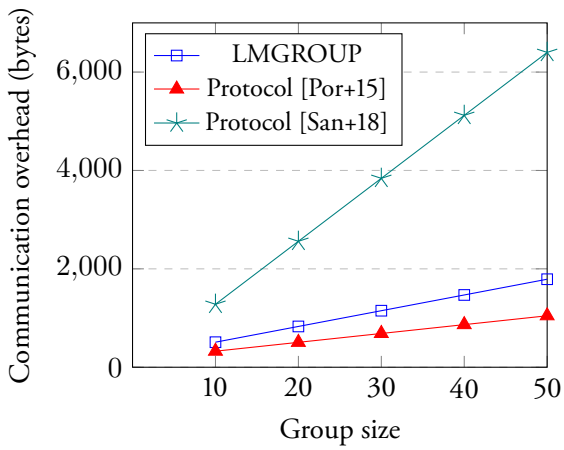


Figure 5: Communication overhead comparison of LMGROUP with protocols [Por+15; San+18]

overhead than SHA256 which was used in LMGROUP. The protocol [San+18] is not a multicast protocol, and as can be seen, it has the highest byte overhead; increasing the group size will cause a significant increase in its communication overhead.

5.2 Storage Overhead

In LMGROUP the following information needs to be stored on the device side: the public key of the server, the device private and public keys, the device *ID*, and some other variables regarding the used ECC curve. We measured the memory footprints of our multicast key establishment scheme using ESP32-S2, and the results are shown in Table 3. DRAM specifies the RAM usage, which is assigned to zero and non-zero values at the program's startup. IRAM indicates the total executable code which is executed from IRAM, and D/IRAM specifies the total size of DRAM and IRAM together. Flash code specifies the total size of executable code which is executed from the flash cache or IROM. Flash rodata, on the other hand, indicates the total size of read-only data that is loaded from the flash cache

Table 3: Memory footprints of LMGROUP

	D/IRAM(B)	Flash Code(B)	Flash rodata(B)	Total image size(B)
LMGROUP	110855	467427	102736	681018

or DROM. Finally, the total image size indicates the estimated total binary file size of the program, which includes the whole size of all used memory types. As indicated in Table 3, the RAM and ROM usage of LMGROUP are reasonable on the IoT device side. Considering the limitations of resource-constrained devices, LMGROUP is applicable to be used in such devices.

6 Formal Security Verification

In order to formally verify the security properties of LMGROUP, we have used ProVerif [Bla+18]. ProVerif uses Dolev-Yao model [DY81] for the adversary model, and it can be used to verify the security properties of cryptographic protocols formally. Applied pi calculus [RS11] is used in ProVerif as the modeling language. In our protocol modeling, we start with the declaration phase, where different protocol components, including variables, functions, and channels, are declared. We used different types in our ProVerif model to declare the type of variables, such as key, nonce (used for session *ID*), point (used for ECC point), and bitstring. The term [private] in front of some variable definitions indicates that those variables are not known by the attacker. Our modeled protocol using ProVerif is also available[4].

The main functions used in the modeling of our key establishment scheme are hash function, ECC point multiplication and addition, MAC, and XOR; these functions are modeled as follows:

```
fun hash(bitstring):bitstring.
fun mul(bitstring,bitstring):bitstring.
fun add(bitstring,point):point.
fun mac(key,nonce,bitstring,point,bitstring) :
bitstring.
fun xor(bitstring,bitstring):bitstring.
```

The functions can have different input and output types; as an example, the MAC function takes five inputs of type key, nonce, bitstring, point, and bitstring, and it generates an output of type bitstring.

After declaring variables and functions, we defined different queries; these queries are used to check whether the protocol has specific security properties or

[4]https://github.com/pegahnikbakht/Multi-key-share/

not. We verified security properties, including secrecy, authentication, and correspondence, through different queries. These security properties can protect against various attacks, including 1) Man in The Middle attack, 2) Replay and Impersonation attacks, and 3) Denial Of Service attack. The queries used to verify the secrecy properties to protect against these attacks are described below.

6.1 Man in The Middle Attack Protection

Man in The Middle Attack (MITM) can be protected through secrecy property. To verify secrecy, we have used the following queries:

```
query attacker ( da ).
query attacker ( db ).
```

In our ProVerif model, we modeled two devices, Device A and B, modeled as *Da* and *Db*, and a server modeled as *S*. In the above queries, *da* and *db* represent the private keys of devices A and B, and the queries check whether the attacker can gain any knowledge about the private keys of those devices or not. ProVerif verification results indicate that the above queries are successfully verified, and the attacker can not get access to the private keys (*da* and *db*) and can not access the information required to generate the group key.

6.2 Replay and Impersonation Attacks Protection

We have used correspondence property to verify protection against replay and impersonation attacks. To prove correspondence, we have used these queries:

```
query a:bitstring,b:nonce,c:bitstring,d:point,
e:bitstring;
event(termDevice(a,b,c,d,e))==>
event(initserver(a,b,c,d,e)).

query a:bitstring,b:bitstring;
event(termserver(a,b))==>event(initDevice(a,b)).
```

As it can be seen, four different events are used in the above queries: *initDevice* and *termDevice* refer to initiating and terminating the device, respectively, and *initserver* and *termserver* refer to initiating and terminating the server. The inputs of the events in the first query represent HMAC, *Session ID*, *Auth*, *R*, and *Se* and the inputs of events in the second query are device *ID* and acknowledgement. The above queries are satisfied if for each occurrence of the event *termDevice*, there is a previous execution of *initserver*, and also if for each occurrence of *termserver* there is a previous execution of *initDevice*. These correspondence relations protect against replay and impersonation attacks. The above queries are successfully verified in ProVerif.

6.3 Denial Of Service Attack Protection

To protect against DoS attacks, we have used the authentication property; if all of the messages are authenticated in the protocol, then it can protect against DoS attacks; we have used *HMAC* verification to verify the authenticity of the messages in LMGROUP. To prove the authentication, again, the correspondence assertion (the relationships between the execution of events) is used in ProVerif, and the same queries used in Section 6.2 are used to verify the authentication property. ProVerif has already verified the correspondence successfully.

7 Conclusion

In this paper, we propose LMGROUP a lightweight and multicast group key establishment scheme for IoT networks. In LMGROUP the IoT devices do not need to interact with each other to gain the shared key; instead, a central server is used to select the group members and send the group information to the members (having some redundant servers is preferred to avoid a single point of failure). In this paper, we suggest an unpredictable group member selection algorithm based on the criticality level (which is decided by the network administrator) of the devices in the network. LMGROUP uses ECIES based operations to share the group key and uses HMAC verification instead of heavy signature verification to authenticate the group key establishment messages. The evaluation result of implementing LMGROUP on a real testbed setup indicates it is lightweight and scalable and can be used in small to large-size networks. The results also indicate that LMGROUP has low storage, communication, and computation costs. Furthermore, we formally verify LMGROUP and prove it is secure and robust against different attacks, including replay and DoS attacks.

Acknowledgments

This work was financially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP).

References

[Beh+19] T. M. Behera et al. “Residual energy-based cluster-head selection in WSNs for IoT application”. In: *IEEE Internet of Things Journal* 6.3 (2019), pp. 5132–5139.

[Bjo09] K. Bjoernsen. “Koblitz Curves and its practical uses in Bitcoin security”. In: *order (ε (GF (2k)* 2.1 (2009), p. 7.

[Bla+18] B. Blanchet et al. “ProVerif 2.00: automatic cryptographic protocol verifier, user manual and tutorial”. In: *Version from* (2018), pp. 05–16.

[Che16] O. Cheikhrouhou. "Secure group communication in wireless sensor networks: a survey". In: *Journal of Network and Computer Applications* 61 (2016), pp. 115–132.

[CK16] Y. Chen and T. Kunz. "Performance evaluation of IoT protocols under a constrained wireless access network". In: *2016 International Conference on Selected Topics in Mobile & Wireless Networking (MoWNeT)*. IEEE. 2016, pp. 1–7.

[Dam+20] M. Dammak et al. "Decentralized Lightweight Group Key Management for Dynamic Access Control in IoT Environments". In: *IEEE Transactions on Network and Service Management* 17.3 (2020), pp. 1742–1757.

[DS06] G. Dini and I. M. Savino. "S2rp: a secure and scalable rekeying protocol for wireless sensor networks". In: *2006 IEEE International Conference on Mobile Ad Hoc and Sensor Systems*. IEEE. 2006, pp. 457–466.

[DY81] D. Dolev and A. C. Yao. "On the Security of Public Key Protocols". In: *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*. SFCS '81. Washington, DC, USA: IEEE Computer Society, 1981, pp. 350–357.

[Fer+18] N. Ferrari et al. "Lightweight group-key establishment protocol for IoT devices: Implementation and performance Analyses". In: *2018 Fifth International Conference on Internet of Things: Systems, Management and Security*. IEEE. 2018, pp. 31–37.

[HD16] F. Hamzeloei and M. K. Dermany. "A TOPSIS based cluster head selection for wireless sensor network". In: *Procedia Computer Science* 98 (2016), pp. 8–15.

[HL10] L. Harn and C. Lin. "Authenticated group key transfer protocol based on secret sharing". In: *IEEE transactions on computers* 59.6 (2010), pp. 842–846.

[Kaz+11] F. Kazemeyni et al. "Group selection by nodes in wireless sensor networks using coalitional game theory". In: *2011 16th IEEE International Conference on Engineering of Complex Computer Systems*. IEEE. 2011, pp. 253–262.

[Kim+16] J. Y. Kim et al. "Seda: Secure over-the-air code dissemination protocol for the internet of things". In: *IEEE Transactions on Dependable and Secure Computing* 15.6 (2016), pp. 1041–1054.

[Lee+11] C.-Y. Lee et al. "Secure key transfer protocol based on secret sharing for group communications". In: *IEICE TRANSACTIONS on Information and Systems* 94.11 (2011), pp. 2069–2076.

[Per+02b] A. Perrig et al. "The TESLA broadcast authentication protocol". In: *Rsa Cryptobytes* 5.2 (2002), pp. 2–13.

[Por+15] P. Porambage et al. "Group key establishment for enabling secure multicast communication in wireless sensor networks deployed for IoT applications". In: *IEEE Access* 3 (2015), pp. 1503–1511.

[PPW20] A. Piccoli, M.-O. Pahl, and L. Wüstrich. "Group Key Management in constrained IoT Settings". In: *2020 IEEE Symposium on Computers and Communications (ISCC)*. IEEE. 2020, pp. 1–6.

[RPC18] T. P. Raptis, A. Passarella, and M. Conti. "Performance analysis of latency-aware data management in industrial IoT networks". In: *Sensors* 18.8 (2018), p. 2611.

[RS11] M. D. Ryan and B. Smyth. "Applied pi calculus". In: *Formal Models and Techniques for Analyzing Security Protocols*. Ios Press, 2011, pp. 112–142.

[Sam16] S. S. I. Samuel. "A review of connectivity challenges in IoT-smart home". In: *2016 3rd MEC International conference on big data and smart city (ICBDSC)*. IEEE. 2016, pp. 1–4.

[San+18] A. S. Sani et al. "A lightweight security and privacy-enhancing key establishment for internet of things applications". In: *2018 IEEE International Conference on Communications (ICC)*. IEEE. 2018, pp. 1–6.

[SLS10] J.-H. Son, J.-S. Lee, and S.-W. Seo. "Topological key hierarchy for energy-efficient group key management in wireless sensor networks". In: *Wireless personal communications* 52.2 (2010), pp. 359–382.

[WGL00] C. K. Wong, M. Gouda, and S. S. Lam. "Secure group communications using key graphs". In: *IEEE/ACM transactions on networking* 8.1 (2000), pp. 16–30.

[ZSJ06] S. Zhu, S. Setia, and S. Jajodia. "LEAP+ Efficient security mechanisms for large-scale distributed sensor networks". In: *ACM Transactions on Sensor Networks (TOSN)* 2.4 (2006), pp. 500–528.

Flowrider: Fast On-Demand Key Provisioning for Cloud Networks

Abstract

Increasingly fine-grained cloud billing creates incentives to review the software execution footprint in virtual environments. For example, virtual execution environments move towards lower overhead: from virtual machines to containers, unikernels, and serverless cloud computing. However, the execution footprint of security components in virtualized environments has either remained the same or even increased. We present Flowrider, a novel key provisioning mechanism for cloud networks that unlocks scalable use of symmetric keys and significantly reduces the related computational load on network endpoints. We describe the application of Flowrider to common transport security protocols, the results of its formal verification, and its prototype implementation. Our evaluation shows that Florwider uses up to an order of magnitude less CPU to establish a TLS session while preventing by construction some known attacks.

1 Introduction

Throughout the past decade, cloud computing has evolved to support a panoply of orchestration, deployment, and billing approaches. Notable trends are the use

Nicolae Paladi, Tiloca, Marco, Pegah Nikbakht Bideh, and Martin Hell. "Flowrider: Fast On-Demand Key Provisioning for Cloud Networks". In *the 17th EAI International Conference on Security and Privacy in Communication Networks, EAI SecureComm 2021, Canterbury, Great Britain(online)*, pp. 207-228. Springer, Cham.

of resource description templates [D W17], emergence of serverless computing [I B17] and fine-grained resource billing [K T20; Y Z17]. Resource description templates allow to dynamically deploy workloads and provision them with cryptographic material or network and application configuration. Most major cloud providers offer serverless computing[1]. This defers the operation of the server platform to the cloud provider, while allowing developers to focus on the application code. Cloud users are billed for the number of function invocations and consumed computation resources, rather than for a pre-purchased unit of computation such as a bare-metal server or a virtual machine. Finally, serverless plans are billed based on the CPU, memory, and I/O operations that functions consume. Fine-grained billing provides strong incentives to develop and deploy applications that utilize a minimum amount of computing resources. This calls for a rigorous review of software development and deployment approaches to reduce the use of computing resources.

Consider software-defined networking (SDN): separation of control and data planes helps network configuration and management; however, network operations security did not keep up with new capabilities enabled by SDN. So far, the distribution of cryptographic material to network endpoints leverage to a limited extent the logical centralization of network control [WZN19]. As a result, public-key cryptography, rather than symmetric key cryptography, remains almost pervasively the tool of choice for enabling secure network traffic in virtualized deployments regardless of the network architecture. While public-key cryptography is robust and scalable, it introduces key management complexity and is relatively CPU-expensive; with fine-grained billing in place, this directly translates into additional financial costs. On virtualized hosts where tenants share a common entropy pool, generating asymmetric keys may slow down applications if sufficient entropy is not available [I D13]. Some network endpoints may even lack the computational capacity to generate cryptographic material without disrupting their own operations. Generating keys on a dedicated host with deep entropy pools can reduce the key creation overhead. On the other hand, while generating symmetric keys requires less computational power and has firmware support on many platforms, the use of symmetric keys leads to challenges such as secure key provisioning and key authentication. This introduces the research question: *can the SDN model be leveraged to conveniently provision symmetric keys and reduce computational resource consumption?*

We posit that the answer is *yes* and demonstrate this with Flowrider, a novel key provisioning mechanism for network endpoints in SDN deployments that considers the practicalities of cloud systems deployment. In particular, Flowrider takes a reactive, on-demand, and automatic approach that embeds key distribution into the network flow establishment. Furthermore, Flowrider makes key distribution agnostic of the network topology and communication patterns in the system, of

[1] See Amazon Lambda, Google Cloud Functions, Azure Functions, Salesforce Evergreen, etc.

which it does not require any early knowledge. Overall, Flowrider reduces the number of steps for providing symmetric key material to endpoints and the time required to set up secure communication.

By conveniently enabling the use of symmetric keys [SPT20], Flowrider reduces by an order of magnitude the computation load of secure channel establishment on network endpoints and simplifies key management in SDN deployments [Pal+21], without compromising communication security properties. Minor modifications of network endpoints introduce another contribution - *flow-specific symmetric keys* - that enable per-flow cryptographic isolation of network traffic. Flowrider is compatible with common transport layer security protocol suites including (D)TLS v1.2 [DR08b; RM12] and v1.3 [Eri18; E R21]. Our contribution is three-fold:

- We describe a key provisioning mechanism that leverages the use of symmetric encryption keys in virtualized deployments within an administrative domain.
- We describe the mechanism and functioning of *flow-specific symmetric keys*, for establishing secure channels between network endpoints.
- We detail how the proposed mechanism works in the (D)TLS security suites v1.2 and v1.3, where it also prevents the "Selfie" attack [DG19] by construction.

Our Flowrider implementation shows reduced computation effort and fewer round-trips to generate authentication credentials and establish secure communication between endpoints. Note that Flowrider primarily targets controlled enterprise environments and does not focus on privacy for network endpoints.

The rest of this paper is organized as follows. We introduce the necessary background in Section 2 and describe the system model, threat model, and assumptions in Section 3. In Section 4, we introduce the Flowrider key provisioning mechanism. In Section 5, we describe the use of Flowrider with (D)TLS. In Section 6, we provide a formal security analysis of Flowrider with ProVerif, followed by an experimental evaluation in Section 7. We review the related work in Section 8 and conclude in Section 9.

2 Background

We first introduce the main concepts and context considered in the rest of the paper.

2.1 Deployment in Virtualized Environments

In modern distributed systems, workloads are commonly deployed using a resource orchestration system such as Kubernetes [E A15], Micado [K T19] or Ranch-

er[2]. Workloads are deployed based on a resource description expressed in a template encoded in a domain-specific language such as TOSCA [Ts 14]. Based on the deployment template, an orchestrator creates and configures workload environments (virtual machine images, containers, or microservices), and deploys them on the underlying hardware. The orchestrator also deploys network components - such as the network controller and network functions - and implements a network configuration defining the communication topology between workload containers. Depending on operating considerations, the orchestrator may be co-located with the network controller. Orchestrators commonly maintain a control channel to patch and update the workloads, re-provision cryptographic material, and collect operation logs. Finally, deployments can be dynamically reconfigured depending on the availability of resources (such as memory, CPU, IO, and bandwidth).

2.2 SDN and OpenFlow

SDN emerged in response to the increasing complexity of network deployments, facilitating operation and management of virtualized networks [A G05]. Its operational advantages lead to wide adoption in enterprise deployments [S J13]. We next introduce several relevant components of the SDN model.

The *data plane* contains hardware and software routing components and implements routing policies that satisfy network administrator goals. It is optimized for forwarding speed but may contain logic for in-network processing [R B16]. The *Southbound API* is a vendor-agnostic set of instructions implemented by the data plane, allowing two-way communication between the data and the control planes. In this paper, we consider the OpenFlow protocol [Ope15]. The *control plane* is an abstraction layer transforming high-level network operator goals into discrete routing policies based on a global network view. *Network functions* are used by network administrators to express their network configuration goals using a set of high-level commands. Examples of such applications are firewalls, intrusion detection systems, traffic shapers, etc. In this paper, we use a custom network function to generate symmetric keys for establishing secure channels between network endpoints.

2.3 Secure Channels

To establish a secure channel, two parties authenticate with each other and derive key material to protect their communication. To this end, two fundamental approaches exist.

The first approach uses a symmetric *pre-shared key* held by both parties. Advantages of this approach include the typically small size of keys, computationally efficient operations needed to use those keys, and resilience against cryptanalysis using quantum-based algorithms. On the other hand, pre-shared keys are typically

[2]https://rancher.com/

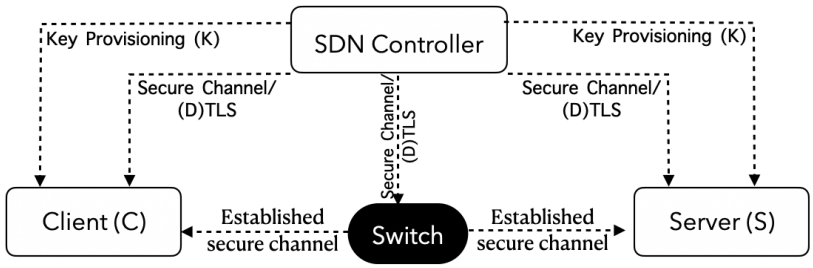


Figure 1: High-level system architecture and components

more difficult to manage, requiring dedicated management procedures to provide, distribute, and revoke them. Management tasks become especially complicated in large and dynamic systems.

The second approach is based on public-key cryptography, where each party acquires the other's public share of a key pair. In practice, this is a bare *raw public key*, or a public *certificate* including the public key and signed by a trusted certification authority. This approach is widely used: since only public information is shared, management tasks are simpler compared to pre-shared keys and can be automated through dedicated Public Key Infrastructure. On the other hand, this approach results in much larger key material, heavier computation load when performing cryptographic operations, and higher entropy requirements on the communication parties.

3 Network Scenario

Consider the network scenario illustrated in Figure 1: an orchestration node collocated on a network controller deploys two endpoints, i.e. C as Client and S as Server, as well as an OpenFlow Switch on the communication path between C and S. Also, it configures the network controller to establish and manage the network flows between the endpoints. For monitoring and patch management purposes, the orchestrator node establishes at deployment time and maintains a secure channel with the endpoints. Note that this approach is in-line with the industry best-practice recommendations [Eur18].

We assume that the network controller established at deployment time three secure communication channels: with C, with S, and with the Switch. These can practically be enforced through (D)TLS sessions. The Switch is able to forward network traffic between C and S, according to the established flows.

For simplicity and with no loss of generality, we hereafter focus on the scenario in Figure 1. Nevertheless, the solution presented in this paper seamlessly works also in more complex and scalable scenarios, where multiple switches, as well as multiple pairs of client and server peers, are deployed. We assume that the network

deployment follows best practices in terms of capacity for network flows, flow establishment rate, and the number of peers engaged in acceptable traffic shapes. This includes proper allocation of bandwidth resources and a sufficient number of deployed switches to prevent bottleneck points and congestion.

With reference to Figure 1, C intends to securely communicate with S. As discussed in Section 2.3, typical approaches to establish a secure communication channel rely on either: a symmetric key pre-shared between C and S; or asymmetric key material either pre-provisioned at orchestration time, exchanged during the secure channel establishment, or acquired out-of-band, such as through a custom PKI infrastructure. We argue that, currently, the above approaches display at least the following limitations.

First, if C and S use multiple network flows, communications on each network flow occur over secure channels created with the same pairwise set of key material. Thus, compromising the single set of key material leads to endangering the data security on all network flows between the two endpoints.

Second, asymmetric key material, e.g. raw public keys and public certificates require computationally- and resource-demanding operations on the endpoints. This becomes critical in virtualized environments and serverless model with fine-grained resource billing and limited entropy pools.

Third, while use of symmetric keys is computationally lightweight and faster than public-key approaches, they are rarely used to establish secure communication between endpoints due to constraints in key provisioning and management. Symmetric keys are harder to distribute and revoke, especially in large-scale and dynamic distributed workload deployments.

Fourth, provisioning of symmetric key material must occur before communication between the endpoints can start. Moreover, it requires pre-knowledge of the network topology and of the communication patterns expected from the two endpoints, further complicating the management of symmetric key material. We describe an alternative solution allowing to: (i) provide per-flow key material, where a single key compromise does not affect the security of other flows; (ii) distribute symmetric key material in a way that is fast, dynamic, and automatic. This approach does not require a priori knowledge of the network topology and communication patterns among the involved endpoints; (iii) facilitate centralized maintenance of software and hardware for cryptographic operations and key generation.

Flowrider achieves this by provisioning the Client and Server with a *flow-specific* symmetric key used to establish a secure communication channel. Key provisioning is done *at flow installation time*, whenever the Client initiates a new communication session with the Server. This approach, further described in Section 4, can be used with various protocol suites for secure channel establishment. In Section 5, we additionally detail how it can be implemented in the (D)TLS suite without transcending the isolation between the transport and encryption layers of a communication session.

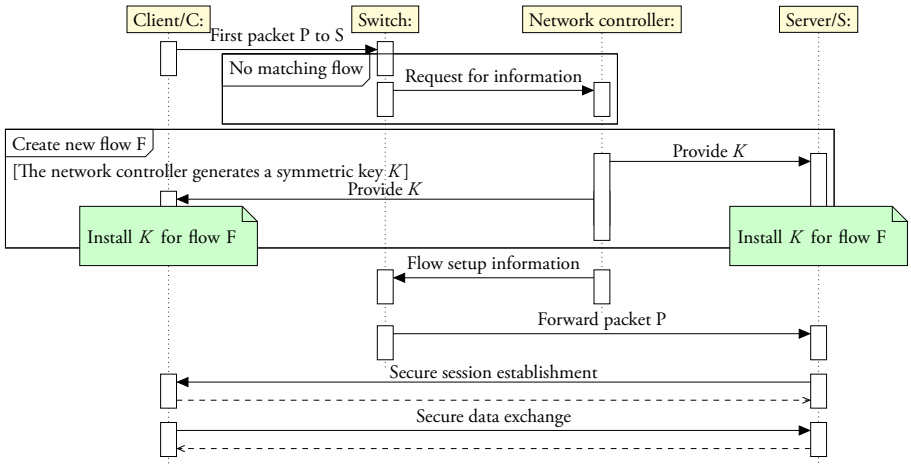


Figure 2: Step-by-step general execution

4 Key Provisioning Method

We next describe *Flowrider*, a novel key distribution method for cloud networks. In particular, Flowrider enables fast, automatic on-demand provisioning of symmetric pre-shared keys to peer endpoints. Pre-shared keys are distributed contextually with the establishment of a network flow between two endpoints and is associated with that respective network flow. Once received, the endpoints can use the pre-shared key to establish a secure channel for communicating over that network flow.

Flowrider builds on the following rationale: each time the Client initiates a session with the Server and triggers the establishment of a new network flow, the network controller generates a new symmetric pre-shared key associated to that flow, and provisions it to both endpoints. To convey the concept, in this paper we assume that the control plane operates in a reactive mode. However, this is not a hard requirement: packets can be matched on the switch while matching packets can be mirrored and upstreamed to the controller.

In the network scenario illustrated in Figure 1, the network Controller provides the Client and the Server with symmetric per-flow keys. Key provisioning is done over the secure channel between the Controller and the Client (C) and Server (S), pre-established at deployment time. Key provisioning is contextual to establishment of a new network flow between C and S, involving the Switch and the Controller.

We illustrate a run-through of Flowrider in Figure 2, with the following steps:

1. C sends the first packet *P* addressed to S. The packet reaches the Switch.
2. The Switch does not find in its flow table a flow rule matching with packet *P*.

3. The Switch sends a control message to the network controller.
4. The network controller:
   (a) Generates a flow rule F to handle traffic between C and S matching packet *P*.
   (b) Generates a cryptographic symmetric key K associated to F, together with a related key identifier[3].
5. The network controller provisions the key K and the related key identifier to both C and S, through the respective pre-established secure channel. The network controller may additionally provide C with the IP address of S, echoing what is specified in the control message from the Switch.
6. C and S install the received key K and related key identifier. If the message from the controller includes also an IP address, C verifies that to be the destination address of its original request to S. This prevents possible internal adversaries from carrying out misbinding attacks based on IP-spoofing.
7. The network controller communicates to the Switch the new flow rule F.
8. The Switch forwards the packet P to S, according to the flow rule F.
9. C and S use the key K to establish a secure session, for example using the (D)TLS Handshake protocol (see Section 5).
10. C and S use the flow F to exchange packets over the established secure channel.

Note that steps 5 and 7 occur concurrently.

4.1 Discussion

In Flowrider, the network controller distributes symmetric keys ad-hoc and on-demand when installing network flows between C and S. Flowrider generates and provisions symmetric keys on a per-flow basis. Hence, different flows between two peer endpoints are related to different and independent security domains. Therefore, compromising the symmetric key associated with a flow does not endanger the security of any other flow between the two endpoints. Note that provisioning symmetric key material is embedded in the OpenFlow control traffic to upstream matching packets and install network flows.

The symmetric key material provided with Flowrider is an alternative to state-of-the-art use of certificates and asymmetric cryptography. Flowrider reduces computational efforts on network endpoints, and hence lowers economic costs. It also

[3]As a possible optimization, the network controller may have generated in advance a number of symmetric keys, which would thus be immediately available to distribute.

reduces entropy requirements for the network endpoints, which is particularly important in virtualized networks.

Section 5 describes how Flowrider can be embodied in versions 1.2 and 1.3 of the security protocol suites TLS and DTLS, without transcending the isolation between the transport layer and (D)TLS. Flowrider is easily and effectively deployable in existing network scenarios that use (D)TLS. Further optimizations are possible, such as indirect provisioning of pre-shared keys to the Server endpoint, through local key derivation on the Server. Section 5.4 describes this optimization with (D)TLS.

The process in Figure 2 refers to a common execution pattern, i.e. where the establishment of the network flow between C and S is triggered by C sending a first packet *P*. Flowrider supports alternative execution patterns, where the SDN deployment is not configured in reactive mode and establishing the network flow - and the consequent key provisioning to C and S - is triggered by the Switch or the network controller, forcing the installation or change of a flow rule. This can happen when enforcing management network policies at deployment time, or when dynamically addressing changes in the network topology and traffic load.

In case of a compromise, the Controller will revoke every flow key issued to a pair of peers. Determining if a peer was compromised can be achieved through intrusion- and anomaly-detection, which are out of the scope of this work. When the Controller determines that one peer P was compromised, the Controller promptly revokes each per-flow key K issued to P which is not yet expired, and notifies any other peer than P that has been provided with K, over the respective secure control channel. This requires the Controller to store at least the key identifier of each non-expired per-flow key.

5 Compatibility with (D)TLS

While Flowrider can be used with various common transport security protocols, we next discuss compatibility with the TLS and DTLS security suites. In Sections 5.2 and 5.3, we describe the embodiment in version 1.2 and 1.3 of (D)TLS, allowing Flowrider to be immediately deployable without breaking existing security standards.

5.1 Transport Layer Security

Most of the network traffic exchanged today, especially on the Internet, is protected at the transport layer. That is, two communicating peers establish a secure channel, namely *session*, and use it to secure the entire application message. The protected message is then handed over to the transport layer, e.g. to the TCP or UDP protocol, for delivery to the other peer. Such secure communication is typically achieved using the protocol suites *Transport Layer Security* (TLS) and *Datagram Transport Layer Security* (DTLS).

The TLS 1.2 protocol suite [DR08b] secures the exchange of application data over TCP among two peers, namely *Client* and *Server*, by preventing the eavesdropping, tampering, and forgery of exchanged messages. The two main protocols composing the TLS suite are the *Handshake* protocol and the encapsulation *Record* protocol. The Client initiates the Handshake execution with the Server, by sending a *ClientHello* Handshake message. Following the Handshake protocol, the two peers agree on a number of security parameters and establish key material to later secure their communications.

The Handshake execution is fundamentally based on two possible approaches, depending on the type of security material pre-installed on the two peers and used during the secure session establishment. In the first approach, the two peers own one or more symmetric pre-shared keys [ET05b], and the Client can suggest to the Server which key it intends to use during the Handshake. In the second approach, the peers rely on asymmetric key pairs, and public keys are exchanged either as conveyed in public certificates [Coo+08] or as raw public keys [Wou+14] generated by manufacturers and installed on nodes before deployment. A node must use out-of-band means for validating raw public keys received from other peers, and usually retains a list of trusted peer identities. Upon successful Handshake completion, peers can exchange application data messages over the established secure session, using the Record protocol.

The DTLS 1.2 protocol suite [RM12] provides secure communication of application data over unreliable datagram protocols such as UDP. DTLS is based on TLS, provides equivalent security guarantees, and relies on analogous Handshake and Record protocols. The DTLS protocol suite has several differences from TLS, to deal with the unreliable underlying datagram transport protocols it runs on. In particular, it does not support stream ciphers, admits preserving secure sessions upon silently discarding invalid incoming messages, and includes an explicit fresh sequence number in every protected message. This allows to correctly distinguish and process incoming DTLS messages, also in case of out-of-sequence delivery due to the unreliable transport service.

Finally, DTLS introduces an optional additional exchange of a stateless *Cookie* between the Client and Server, as a first step of the Handshake. Upon receiving a first ClientHello message, the Server can reply with a *HelloVerifyRequest* message, including a locally generated value as Cookie. The Client must then reply by sending a second ClientHello, which includes the same Cookie. The Handshake further continues only if the Server successfully verifies the Cookie received in this second ClientHello. This forces the Client to prove its alleged source IP address, and, possibly in combination with additional means such as [M T17], complicates possible Denial of Service attacks against the Server performed by an active adversary able to spoof IP addresses.

TLS 1.3 was released to improve both performance and security assurances [Eri18]. While fundamentally providing the same security guarantees as TLS 1.2, TLS 1.3: i) reduces the handshake by one round trip, while having more handshake

messages also encrypted; ii) provides new functions for key material derivation, with improved key separation and facilitating cryptographic analysis; iii) always provides perfect forward secrecy if peers run the handshake through public-key based key establishment; iv) supports the latest key establishment, cipher, and signature algorithms, deprecating insecure or obsolete ones; and v) enables the exchange of early secure data at the beginning of the handshake, at the cost of sacrificing a subset of security properties for such data. While TLS 1.3 has been increasingly adopted since its release, TLS 1.2 is expected to continue being used for a long time, as (a dominant) protocol suite for secure communication.

5.2 Flowrider with (D)TLS 1.2

Assume the Client and Server intend to securely communicate using the TLS 1.2 [DR08b] or DTLS 1.2 [RM12] protocol suite. With reference to the steps in Section 4 shown in Figure 2, Flowrider can be embedded in the (D)TLS Handshake protocol as follows.

At Step (1), the first packet P from C addressed to S is either a TCP SYN (for a TLS handshake) or a *ClientHello* Handshake message (for a DTLS handshake). In either case, C performs the (D)TLS Handshake with S in pre-shared key mode [ET05b].

Later on during the Handshake execution, i.e. at Step (9) of the Flowrider execution (see Section 4), C points S to key K to be used as a pre-shared key for mutual authentication and as input for deriving the (D)TLS session key material. C specifies the key identifier of the key K in the *PSK identity* field of the *ClientKeyExchange* Handshake message sent to S.

5.3 Flowrider with (D)TLS 1.3

Assume that the Client and Server intend to securely communicate using the TLS 1.3 [Eri18] or DTLS 1.3 [E R21] protocol suite. Flowrider can be embedded in the (D)TLS Handshake protocol as follows (see Section 4, Figure 2).

At Step (1), the first packet P from C addressed to S is either a TCP SYN (for a TLS handshake) or a *ClientHello* Handshake message (for a DTLS handshake). In either case, C performs the (D)TLS Handshake with S in pre-shared key mode. That is, as per (D)TLS 1.3 [E R21][Eri18], C has to include in the *ClientHello* Handshake message:

1. A *psk_key_exchange_modes* ClientHello extension, which specifies the *psk_ke* or *psk_dhe_ke* key exchange mode.

2. A *pre_shared_key* ClientHello extension, present as the last extension and including a collection of offered pre-shared keys. This collection is structured as follows: (1) a list of key identifiers; (2) a list of *key binders*, one for each pre-shared key and in the same order as the key identifier list. Each key

binder is an HMAC computed with a binder key derived from the corresponding pre-shared key. The key binder is computed over the *ClientHello* message up to and including the key identifier list of the *pre_shared_key* ClientHello extension.

S expects a valid hint of the pre-shared key already at the first (D)TLS *ClientHello* message. However, if DTLS is used, C does not have the key K and its key identifier from the network controller already at Step (1) of the Flowrider execution, where the first packet P addressed to S is already the *ClientHello* message. Thus, when starting a new communication flow in the DTLS case, the Client cannot produce a *ClientHello* message, as the per-flow symmetric key is not available yet. Intuitively, this is overcome by the network controller finalizing the original and incomplete *ClientHello* message, before the Switch eventually forwards it to the Server as per the newly established traffic flow. In particular, C stores a dummy pre-shared symmetric key and a related key identifier, which is not associated with any corresponding server. Then, the following adaptation of the Flowrider execution is performed, as also shown in Figure 3.

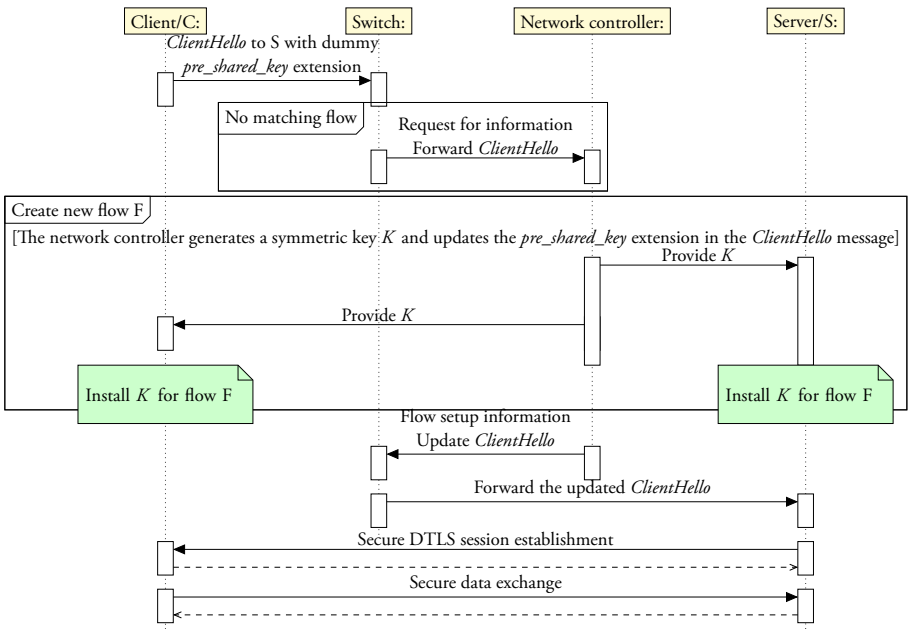


Figure 3: Step-by-step execution for implementing Flowrider in DTLS 1.3

1. C sends the *ClientHello* message in the first packet P addressed to S. In particular, the *pre_shared_key* ClientHello extension offers only the dummy pre-shared key used by C for this purpose. Then, the packet reaches the Switch.

2. The Switch fails to find in its flow table a flow rule matching with packet *P*.
3. The Switch sends a control message to the network controller, asking for information about setting up a new flow between C and S and also forwards the entire packet P, including *ClientHello*, to the network controller.
4. The network controller:
   (a) generates a new flow rule F to handle traffic between C and S akin to packet *P*;
   (b) generates a cryptographic symmetric key K associated to flow F, together with a related key identifier;
   (c) builds a new *pre_shared_key* ClientHello extension for the *ClientHello* message in the packet *P*. The new extension offers only the key K associated to flow F, and includes one consistently recomputed key binder. The recomputed extension replaces the one originally included in the *ClientHello* message in the packet *P*.
5. The network controller provisions the key K and the related key identifier to both C and S, through the respective pre-established secure channel.
6. Both C and S install key K and related key identifier.
7. The network controller replies to the Switch with:
   (a) information on handling packets in the new flow F;
   (b) packet *P* including the updated *ClientHello* message.
8. The Switch forwards the packet P to S, as per the newly installed flow F.
9. C and S establish a secure session/channel, by using the key K, as per the DTLS 1.3 Handhshake protocol.
10. C and S use the flow F to exchange packets over the established DTLS 1.3 channel.

5.4 Optimization through key derivation

As an optimization, the network controller may not explicitly provide S with the key K. Instead, S can *derive* the key K from its key identifier, provided by C as a hint during the (D)TLS Handshake, allowing to further reduce the communication overhead. The optimization requires that:

- The network controller and S share a pairwise symmetric key-derivation key K^*.

- The network controller maintains a counter N_S, which is uniquely associated with S and incremented upon generating a new per-flow key K associated to S.
- The network controller generates the key K by means of a secure key derivation function $PRF(\cdot)$ that takes as input the key-derivation key K^* and a nonce N set as the current value of the counter N_S. $PRF(\cdot)$ can be based on a $HMAC$ function [KBC97b] and rely on the same data expansion scheme described in [DR08b].
- Nonce N used to generate the key K is also used as the key identifier of that key.

In (D)TLS 1.2, the Client C simply specifies the nonce N as a key identifier for the key K in the *PSK identity* field of the *ClientKeyExchange* Handshake message. Upon receiving the *ClientKeyExchange* Handshake message, S derives the key K by means of $PRF(\cdot)$, using the retrieved nonce N and the key-derivation key K^*. This approach was discussed in [G S15].

In (D)TLS 1.3, the nonce N is used as the key identifier for the key K in the *pre_shared_key* ClientHello extension for the *ClientHello* message. In TLS 1.3, this is directly specified by C, after having received the key K from the network controller. In DTLS 1.3, this is specified by the network controller, when building the new *pre_shared_key* ClientHello extension for the *ClientHello* message in the packet P (see step (4c) in Section 5.3). In either case, upon receiving the *ClientHello* message, S derives the key K by means of $PRF(\cdot)$, using the retrieved nonce N and the key-derivation key K^*.

5.5 On Preventing the Selfie Attack

Flowrider prevents the reflection attack ("Selfie" [DG19]) against (D)TLS, which tricks a session peer into processing messages generated by itself, assuming they come from the other peer. This exploits the use of the same pre-shared key in two secure sessions, as (D)TLS client and (D)TLS server.

In an SDN deployment, a peer A (B) acting as (D)TLS client (server) results in one flow, as an exact combination of source address/port and destination address/port. Instead, peer A (B) acting as (D)TLS server (client) results in a different flow, with a flipped combination of source and destination address/port.

In Flowrider, the SDN Controller generates and provides two different pre-shared keys to peers A and B, one for each of the flows. A and B never use the same pre-shared key for both combinations of roles, as they always result in different flows, and distributed pre-shared keys are per-flow. Thus, a given peer gets one different pre-shared key for each role that such peer has with the other peer sharing the same key, and the Selfie attack is prevented by construction.

6 Formal Security Verification

We verified the security properties of Flowrider, using ProVerif [B B20]. ProVerif is based on the *applied pi calculus* modeling language and can represent processes, their interactions, and available security channels. ProVerif considers an active adversary (Dolev-Yao model [D D83]) that cannot decrypt encrypted messages without accessing the secret keys.

6.1 ProVerif Modeling

To model Flowrider with ProVerif, we started by declaring types, cryptographic functions, security assumptions, queries, and processes. Throughout the model, we maintain the assumption of a pre-established secure channel between the network controller and the endpoints (Client and Server), consistently with the network scenario presented in Section 3). The channels were securely established using key material assumed to be inaccessible and infeasible to derive for the adversary. The Client, the Server, the Switch, and the network controller are each modeled as independent, top-level processes.

We verified[4] the following security properties of Flowrider: i) the secure provisioning and resulting secrecy of key *K*, i.e. the key associated with the flow between the Client and Server (see Section 6.2); and ii) the mutual secure possession of key *K* by Client, Server, and controller (see Section 6.3). Note that we *do not* verify security properties that are assumed to be already satisfied, such as the security of the (pre-)established secure sessions and the security of session establishments themselves. In particular, the security of the TLS session establishment has been formally verified in [C C17].

6.2 Key secrecy

In the protocol model we assume that, upon receiving from the controller the key K associated with the flow, C and S use it to derive the key material for the secure session. While this consists of executing a session establishment protocol, the model assumes that a cryptographically secure Key Derivation Function (KDF) is used to derive a single session key K_S as key material. The KDF takes as input the flow key K and context information related to the secure session. Once the Client-Server session is established, the Client sends a message M to the Server, encrypted using the session key K_S. We verified that the adversary cannot access the secret message, with the following query:

$$query(attacker(M)) \tag{1}$$

[4] ProVerif scripts available at `https://anonymous.4open.science/r/8e9da3de-6ccd-4f49-b925-389fbcc9bca6/`

The model successfully verified the secrecy of message M. Since K was used as input to securely derive the session key K_S, in turn used to protect the message M, we conclude that the secrecy of key K is also preserved.

6.3 Mutual secure key possession

In Flowrider, only the Client and Server with access to the flow key K can successfully establish a secure session with each other in a symmetric mode, over that flow. We verify that the parties that possess K can establish a secure session over the flow associated with *K*.

To this end, we verified that, if the Server receives an encrypted message M from the Client over a flow, then i) the Client has previously established a secure session with the Server over that flow; and ii) the Client has sent the message M to the Server, encrypted with the session key K_S derived from the flow key K associated to that flow.

ProVerif allows specifying send and receive operations, as well as to initiate and terminate communications between the Client and Server, by means of events such as Initclient(K_S), Termserver(K_S), Initserver(K_S,Ack) and Termclient(K_S,Ack), where K_S is the session key K_S derived from the flow key *K*. The session establishment is successfully completed by both parties when each of them received an acknowledgment from the other party, over that session. The formalized queries for the above events are:

$$\begin{aligned} & inj-event(termclient(K_s, Ack)) \\ ==> & inj-event(initserver(K_s, Ack)) \end{aligned} \tag{2}$$

$$\begin{aligned} & inj-event(termserver(K_s)) \\ ==> & inj-event(initclient(K_s)) \end{aligned} \tag{3}$$

Queries 2 and 3 verify that for all Initserver(K_S,Ack) and Initclient(K_S), events Termclient(K_S,Ack) and Termserver(K_S) were previously executed. ProVerif successfully verified both correspondence properties in queries 2 and 3. This implies that only the Client possessing the flow key K can connect to the Server over that flow, protecting messages with the session key K_S derived from *K*. This also implies that the Server accepts communications over that flow only from the Client corresponding to the flow key *K*, i.e. exchanging messages encrypted with the session key K_S derived from *K*.

The injective correspondence in query 2 and 3 verifies that the relation between correspondence events is one to one, implying that the Client with access to flow key K can successfully open a dedicated session with the Server. Injectivity holds and ProVerif verified the injective correspondence since the Server should complete

the session establishment using the flow key only once for the session initiated by the Client.

6.4 Verifying the optimization through key derivation

Flowrider can be further optimized for certain (D)TLS protocol use cases (see Section 5.4). In this optimization, the controller does not send the flow key K to the Server. Instead, the Server locally derives the flow key K using a nonce generated by the controller and a long-term symmetric key shared with the controller. The nonce is used as a key identifier for the flow key K and is specified in the session establishment message addressed to the Server. We verified the optimized version of Flowrider and included the nonce in the first message sent out by the Client to the Switch.

$$\begin{aligned} &event(termclient(K_s, Ack)) \\ ==> &event(initserver(K_s, Ack)) \end{aligned} \tag{4}$$

We verified the security properties discussed in Sections 6.2 and 6.3. In this case, we verified only the correspondence, since we considered also multiple flows between the Client and the Server. For non-injective correspondence, the one to one relation between events is not required, but only the event after the arrow is executed prior to the event before the arrow. The formalized queries are:

$$event(termserver(K_s)) ==> event(initclient(K_s)) \tag{5}$$

ProVerif verified the security properties of the optimized Flowrider version.

7 Experimental Evaluation

In order to understand the practical implementation aspects, trade-offs and performance of Flowrider, we implemented[5] it in a distributed virtualized environment. We ran the experiments on Google Compute Platform [S P15] in a `g1-small` virtual machine (VM) instance (1 vCPU, 1.7 GB memory).

The test bed is distributed between four Docker containers with the following roles (see Figure 4): (a) Client, (b) Server, (c) Controller, (d) Open vSwitch (OvS). The endpoints (Client and Server) use TLS 1.3 [Eri18] implemented with the GnuTLS library [N M15], version 3.6.5. Two distinct but closely related Client and Server implementations were created for using symmetric keys and certificates respectively. The controller container runs Ryu 3.12 and a custom Python application, that defines packets to be matched and subsequently generates and delivers keys to the endpoints. The OvS container runs an instance of Open vSwitch that

[5] Implementation code available at `https://anonymous.4open.science/r/8e9da3de-6ccd-4f49-b925-389fbcc9bca6/`

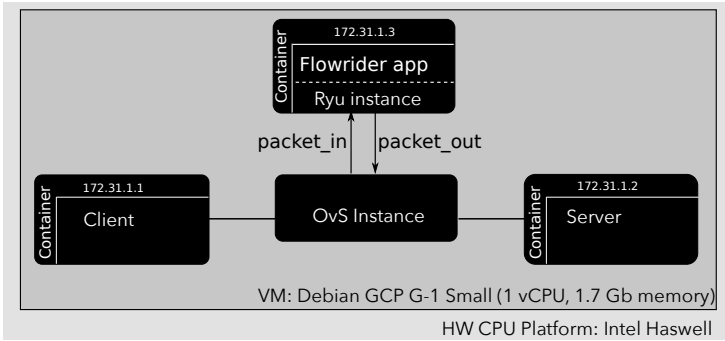


Figure 4: System test bed

routes packets between endpoints and forwards predefined packet types to the controller.

We measured the performance of establishing a TLS session in two scenarios. We ran the TLS handshake in asymmetric mode using PKI certificates (vanilla scenario) and in pre-shared key (PSK) mode using symmetric keys (Flowrider scenario) consistently with the Flowrider embodiment for TLS 1.3 (see Section 5.3). The Client established a TLS session with the Server in the considered mode and terminated the session immediately afterward. We ran the experiment 10,000 times. In both cases, the OvS flow table did not contain any flows between the Client and the Server; as a result, the first Client message (TCP SYN) was forwarded to the controller in each scenario run.

We illustrate the results of our experimental evaluation in Figure 5 and Table 1. Figure 5 shows that the 'PSK' scenario (representing Flowrider) performs better in terms of time spent on the task and CPU utilization. Time elapsed is longer for the 'PSK' scenario, partly due to the overhead introduced by the communication between the switch and the controller. However, the overhead is mostly offset by distributing the pre-shared keys after the first TCP packet, before the TLS session negotiation starts.

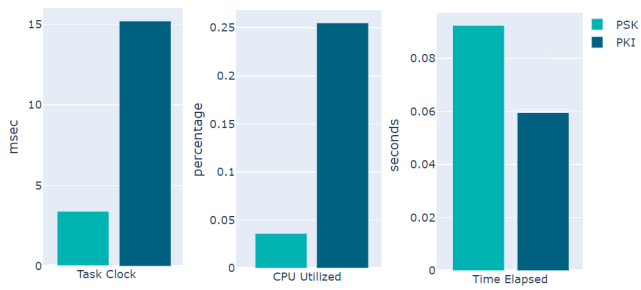


Figure 5: Task Clock, CPU utilisation and Time elapsed for PSK/PKI scenarios

Table 1: Overview of the performance measurements data set

Type	PSK			PKI		
	Task clock, msec	CPU utilized	Time elapsed, msec	Task clock, msec	CPU utilized	Time elapsed, msec
Minimum	3.17	0.034	0.087	13.71	0.183	0.056
Maximum	4.43	0.049	0.097	16.06	0.269	0.080
Mean	3.37	0.037	0.089	14.31	0.24	0.058
Median	3.34	0.037	0.089	14.23	0.24	0.057
Stddev	0.096	0.001	0.0006	0.32	0.005	0.001
Variance	0.00923	0.000001	0.0000004	0.1081	0.00002	0.000002

Table 1 presents a more detailed view. The mean task clock is lower in the Flowrider ('PSK') scenario (3.37 msec compared to 14.31 msec). The CPU utilization is *significantly* lower in the Flowrider ('PSK') scenario, with a mean of 0.037 versus 0.24 CPU. The time elapsed is about 30% higher in the case of the Flowrider ('PSK') scenario. However, considering that this delay occurs *once* at setup time and is not recurrent, we consider that this is an acceptable overhead.

The Flowrider scenario highlights an order of magnitude lower CPU consumption, due to the use of symmetric key material when establishing the TLS session. Note that the overall time to establish the secure channel does not change significantly. In fact, the very first step of the TCP session establishment triggers the distribution of the symmetric key, which is used to establish the TLS session in PSK mode.

8 Related Work

Protocols such as Kerberos [Neu+05] are widely used for symmetric key distribution. This involves a Key Distribution Center - a Trusted Third Party generating and distributing ephemeral keys to clients, without disclosing the secret shared key of the server. Internet applications often rely on an Authorization Server providing trusted assertions to servers about requesting clients [Har12].

Flowrider key distribution can be viewed as a three-party setting, with the switch acting as a relay and middleman. Three-party authenticated key exchange has received much attention. Its security was formalized by Bellare and Rogaway in [M B95] and much research has focused on the password-based variant, introduced in [S B92] and given for the three-party case in [M S95]. In the three-party password-based authenticated key exchange (3PAKE), low entropy secrets shared with the server are used to negotiate a session key between two parties. This protocol in [M S95] was shown to have weaknesses [Y D95; CL 00] and many variants have been proposed since then, some using a server public key [HT 03; TF 09], and some that do not [CL 01; TY 11]. More recent work on PAKE protocols include making them post-quantum secure, both in the two-party setting [J D17] and for three parties [C L19]. While authenticated key exchange protocols assume an unprotected channel and pre-shared keys, Flowrider uses TLS as the underlying protocol for distributing the key from the cluster manager to the involved

parties. This can be accomplished either through symmetric pre-shared keys or public keys and is not predetermined by the Flowrider protocol. Flowrider adapts the problem of three-party key distribution to the SDN setting. The cluster manager is a natural part of the network, and not an otherwise added trusted third party (Key Distribution Center) as in the case with e.g, Kerberos. Since TLS is already used e.g., for deploying jobs to the endpoints and configuring the switch, there is no need to implement additional key exchange protocols. Moreover, new cryptographic primitives incorporated into TLS can be used by Flowrider to take advantage of improved ECC [A L16] and post-quantum resistant algorithms [E C19].

Key distribution for SDN deployments was explored in several contexts. Li *et al.* proposed a symmetric key generation and distribution for content delivery network interconnections using SDN and application-layer traffic optimization [SB09]. The mechanism relies on key generation on the endpoints and a central entity for matching and distributing key pairs. Similar to Flowrider, this relies on a central authority. However, it neither reduces the computational load on the endpoints nor improves the performance of the key exchange. Cloud frameworks commonly rely on a central authority to provision authentication material to virtual instances (either virtual machines or containers) before deployment [D S14; S M15]. Provisioning authentication credentials before instantiation reduces the computational load on the endpoints and reduces the entropy requirements. However, the use of public keys certificates for key establishment requires more round trips compared to protocols using symmetric keys.

Provisioning cryptographic material to network endpoints by storing it in trusted execution environments (TEEs) was explored in both academia and industry [K S16].While this approach leverages hardware security guarantees to store the provisioned cryptographic material, it also introduces additional overhead on accessing the cryptographic material. This includes both provisioning the material to TEEs and retrieving it from TEEs. Finally, other less common approaches rely on information that may be public or not unique, such as the serial number of the device [E M13], or require manual steps that do not scale in production settings [Ope19].

Flowrider builds on earlier work and leverages the OpenFlow protocol to enable symmetric key provisioning. In contrast to existing approaches, Flowrider drastically reduces the computational requirements for supporting end-to-end encryption; it reduces the number of steps for providing symmetric key material to two endpoints and hence for them to set up secure communication; finally, it allows granular cryptographic isolation of network flows. While Flowrider does not require TEE support on network endpoints, it is complementary to approaches provisioning credentials to TEEs.

9 Conclusion

We have presented Flowrider, a novel approach to distribute cryptographic symmetric keys to endpoints in software networks, contextually with network flow establishment. Flowrider efficiently provisions symmetric key material and significantly reduces the number of CPU cycles needed to establish a secure communication channel between two endpoints. Flowrider leverages the logical centralization of software-defined networks to enable efficient use of symmetric keys.

Furthermore, Flowrider makes key distribution agnostic of the network topology and communication patterns in the system, of which it does not require any early knowledge. Finally, Flowrider is compatible with the (D)TLS 1.2 and 1.3 security protocol suites, with only minor modifications to endpoint implementations. Our experimental performance evaluation shows that Flowrider requires up to an order of magnitude less CPU for a TLS session establishment. Future work will focus on the embodiment and evaluation of Flowrider in alternative protocols for secure channel establishment.

Acknowledgments

This work was financially supported in part by the Swedish Foundation for Strategic Research, with the grant RIT17-0035; by the H2020 project SIFIS-Home (Grant agreement 952652); VINNOVA and the CelticNext project CRITISEC and by the Wallenberg AI, Autonomous Systems and Software Program (WASP).

References

[A G05] A. Greenberg and G. Hjalmtysson and D. A. Maltz and A. Myers and J. Rexford and G. Xie and H. Yan and J. Zhan and H. Zhang. “A Clean Slate 4D Approach to Network Control and Management”. In: *SIGCOMM Comput. Commun. Rev.* 35.5 (Oct. 2005), pp. 41–54.

[A L16] A. Langley and M. Hamburg and S. Turner. *Elliptic Curves for Security*. RFC 7748. 2016.

[B B20] B. Blanchet. *ProVerif: Cryptographic protocol verifier in the formal model*. 2020.

[C C17] C. Cremers and M. Horvat and J. Hoyland and S. Scott and T. van der Merwe. “A Comprehensive Symbolic Analysis of TLS 1.3”. In: *Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 1773–1788.

[C L19] C. Liu and Z. Zheng and K. Jia and Q. You. "Provably Secure Three-Party Password-Based Authenticated Key Exchange from RLWE". In: *Information Security Practice and Experience*. Ed. by S.-H. Heng and J. Lopez. Cham: Springer International Publishing, 2019, pp. 56–72.

[CL 00] C.-L. Lin and H.-M. Sun and T. Hwang. "Three-party encrypted key exchange: attacks and a solution". In: *ACM SIGOPS Operating Systems Review* 34.4 (2000), pp. 12–20.

[CL 01] C.-L. Lin and H.-M. Sun and M. Steiner and T. Hwang. "Three-party encrypted key exchange without server public-keys". In: *IEEE Communications letters* 5.12 (2001), pp. 497–499.

[Coo+08] D. Cooper et al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280 (Proposed Standard). RFC. Updated by RFC 6818. Fremont, CA, USA: RFC Editor, 2008.

[D D83] D. Danny and C. Y. Andrew. "On the security of public key protocols". In: *IEEE Trans. on Information Theory* 29.2 (Mar. 1983), pp. 198–208.

[D S14] D. S. Dodgson and R. Farina and J. A Fontana and R. A. Johnson and D. Maw and A. Narisi. *Automated provisioning of virtual machines*. US Patent App. 13/547,148. Jan. 2014.

[D W17] D. Weerasiri and M. C. Barukh and B. Benatallah and Q. Z. Sheng and R. Ranjan. "A Taxonomy and Survey of Cloud Resource Orchestration Techniques". In: *ACM Comput. Surv.* 50.2 (2017).

[DG19] N. Drucker and S. Gueron. *Selfie: reflections on TLS 1.3 with PSK*. Cryptology ePrint Archive, Report 2019/347. `https://eprint.iacr.org/2019/347`. 2019.

[DR08b] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246 (Proposed Standard). RFC. Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919. Fremont, CA, USA: RFC Editor, 2008.

[E A15] E. A. Brewer. "Kubernetes and the Path to Cloud Native". In: *Proc. of the Sixth ACM Symposium on Cloud Computing*. SoCC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 167.

[E C19] E. Crockett and C. Paquin and D. Stebila. “Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH”. In: *NIST 2nd Post-Quantum Cryptography Standardization Conf. 2019*. Gaithersburg, MD, USA: NIST, 2019, pp. 1–24.

[E M13] E. Moret and R. Hubbard and K. A. Watsen and M. Murthy and N. Beauchesne. *Systems and methods for provisioning network devices*. US Patent 8,429,403. Apr. 2013.

[E R21] E. Rescorla and H. Tschofenig and N. Modadugu. *The Datagram Transport Layer Security (DTLS) Protocol Version 1.3*. Internet-Draft. Work in Progress. Apr. 2021.

[Eri18] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. RFC. Fremont, CA, USA: RFC Editor, 2018.

[ET05b] P. Eronen and H. Tschofenig. *Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)*. RFC 4279 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, 2005.

[Eur18] European Telecommunications Standards Institute. *ETSI GS NFV-SEC 014 V3.1.1 (2018-04) - Network Functions Virtualisation (NFV) Release 3; NFV Security; Security Specification for MANO Components and Reference points*. 2018.

[G S15] G. Selander. *WO/2015/002581 Key establishment for constrained resource devices*. 2015.

[Har12] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749 (Proposed Standard). RFC. Updated by RFC 8252. Fremont, CA, USA: RFC Editor, 2012.

[HT 03] H.-T. Yeh and H.-M. Sun and T. Hwang. “Efficient three-party authentication and key agreement protocols resistant to password guessing attacks”. In: *Journal of Information Science and Engineering* 19.6 (2003), pp. 1059–1070.

[I B17] I. Baldini and P. Castro and K. Chang and P. Cheng and S. Fink and V. Ishakian and N. Mitchell and V. Muthusamy and R. Rabbah and A. Slominski and P. Suter. “Serverless Computing: Current Trends and Open Problems”. In: *Research Advances in Cloud Computing*. Singapore: Springer Singapore, 2017, pp. 1–20.

[I D13] I. Damgård and T. P. Jakobsen and J.B. Nielsen and J. I. Pagter. “Secure Key Management in the Cloud”. In: *Cryptography and Coding*. Ed. by M. Stam. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 270–289.

[J D17] J. Ding and S. Alsayigh and J. Lancrenon and RV Saraswathy and M. Snook. "Provably secure password authenticated key exchange based on RLWE for the post-quantum world". In: *Cryptographers' Track at the RSA Conf.* Cham: Springer, 2017, pp. 183–204.

[K S16] K. Sood and J. B. Shaw and J. R. Fastabend. *Technologies for secure inter-virtual network function communication*. US Patent 9,407,612. 2 2016.

[K T19] K. Tamas and P. Kacsuk and J. Kovacs and B. Rakoczi and A. Hajnal and A. Farkas and G. Gesmier and G. Terstyanszky. "MiCADO, a Microservice-based Cloud Application-level Dynamic Orchestrator". In: *Future Generation Computer Systems* 94 (2019), pp. 937–946.

[K T20] K. Thimmaraju and S. Schmid. *Towards Fine-Grained Billing For Cloud Networking*. 2020.

[KBC97b] H. Krawczyk, M. Bellare, and R. Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104 (Informational). RFC. Updated by RFC 6151. Fremont, CA, USA: RFC Editor, 1997.

[M B95] M. Bellare and P. Rogaway. "Provably Secure Session Key Distribution: The Three Party Case". In: *Proc. of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*. STOC '95. New York, NY, USA: Association for Computing Machinery, 1995, pp. 57–66.

[M S95] M. Steiner and G. Tsudik and M. Waidner. "Refinement and extension of encrypted key exchange". In: *ACM SIGOPS Operating Systems Review* 29.3 (1995), pp. 22–30.

[M T17] M. Tiloca and C. Gehrmann and L. Seitz. "On Improving Resistance to Denial of Service and Key Provisioning Scalability of the DTLS Handshake". In: *International Journal of Information Security* 16.2 (Apr. 2017), pp. 173–193.

[N M15] N. Mavrogiannopoulos and S. Josefsson and D. Ueno and C. Latze and A. Pironti and T. Zlatanov and A. McDonald. *GnuTLS Reference Manual*. London: Samurai Media Ltd., 2015.

[Neu+05] C. Neuman et al. *The Kerberos Network Authentication Service (V5)*. RFC 4120 (Proposed Standard). RFC. Updated by RFCs 4537, 5021, 5896, 6111, 6112, 6113, 6649, 6806, 7751, 8062, 8129. Fremont, CA, USA: RFC Editor, 2005.

[Ope15] Open Networking Foundation. *OpenFlow Switch Specification*. Tech. rep. ONF TS-025. v.1.5.1. Open Networking Foundation, Mar. 2015.

[Ope19] Open vSwitch. *Open vSwitch with SSL*. 2019.

[Pal+21] N. Paladi et al. "On-demand Key Distribution for Cloud Networks". In: *2021 24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. 2021, pp. 80–82.

[R B16] R. Bifulco and J. Boite and M. Bouet and F. Schneider. "Improving SDN with InSPired Switches". In: *Proc. of the Symposium on SDN Research*. SOSR '16. New York, NY, USA: ACM, 2016, 11:1–11:12.

[RM12] E. Rescorla and N. Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347 (Proposed Standard). RFC. Updated by RFCs 7507, 7905. Fremont, CA, USA: RFC Editor, 2012.

[S B92] S. Bellovin and M. Merritt. "Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks". In: *Security and Privacy, IEEE Symposium on* 0 (Apr. 1992), p. 72.

[S J13] S. Jain, and A. Kumar and S. Mandal and J. Ong and L. Poutievski and A. Singh and S. Venkata and J. Wanderer and J. Zhou and M. Zhu and J. Zolla and U. Hölzle and S. Stuart and A. Vahdat. "B4: Experience with a Globally-deployed Software Defined WAN". In: *Proc. of the ACM SIGCOMM 2013 Conf. on SIGCOMM*. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 3–14.

[S M15] S. Martinelli and H. Nash and B. Topol. *Identity, Authentication, and Access Management in OpenStack: Implementing and Deploying Keystone*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2015.

[S P15] S. P. T. Krishnan and J. L. U. Gonzalez. "Google Compute Engine". In: *Building Your Next Big Thing with Google Cloud Platform: A Guide for Developers and Enterprise Architects*. Berkeley, CA: Apress, 2015, pp. 53–81.

[SB09] J. Seedorf and E. Burger. *Application-Layer Traffic Optimization (ALTO) Problem Statement*. RFC 5693 (Informational). RFC. Fremont, CA, USA: RFC Editor, 2009.

[SPT20] G. Selander, N. Paladi, and M. Tiloca. *Security for distributed networking*. World Intellectual Property Organization - PCT/EP2019/051456. July 2020.

[TF 09] T.-F. Lee and J.-L. Liu and M.-J. Sung and S.-B. Yang and C.-M. Chen. "Communication-efficient three-party protocols for authentication and key agreement". In: *Computers & Mathematics with Applications* 58.4 (2009), pp. 641–648.

[Ts 14] Ts Binz and U. Breitenbücher and O. Kopp and F. Leymann. "TOSCA: Portable Automated Deployment and Management of Cloud Applications". In: *Advanced Web Services*. New York, NY: Springer New York, 2014, pp. 527–549.

[TY 11] T.-Y. Chang and M.S. Hwang and W.-P. Yang. "A communication-efficient three-party password authenticated key exchange protocol". In: *Information Sciences* 181.1 (2011), pp. 217–226.

[Wou+14] P. Wouters et al. *Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)*. RFC 7250 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, 2014.

[WZN19] H. Wang, Y. Zhao, and A. Nag. "Quantum-Key-Distribution (QKD) Networks Enabled by Software-Defined Networks (SDN)". In: *Applied Sciences* 9.10 (May 2019), p. 2081.

[Y D95] Y. Ding and P. Horster. "Undetectable on-line password guessing attacks". In: *ACM SIGOPS Operating Systems Review* 29.4 (1995), pp. 77–86.

[Y Z17] Y. Zhu and J. Ma and B. An and D. Cao. "Monitoring and Billing of A Lightweight Cloud System Based on Linux Container". In: *2017 IEEE 37th International Conf. on Distributed Computing Systems Workshops (ICDCSW)*. New York, NY, USA: IEEE, 2017, pp. 325–329.

Popular Science Summary in English

Popular Science Summery

You may use various smart devices in your home, such as light bulbs, cameras, etc. These devices are usually connected to the Internet; this makes them accessible anywhere and at any time. You can generally connect to smart devices, for example, through your smartphones, and control them remotely. However, such devices might have security flaws (known as vulnerabilities) that can cause attackers to attack and control them remotely. This situation is undesirable since you do not want an attacker to access your devices or even use them to perform large-scale attacks. Whenever new security vulnerabilities are discovered, the device manufacturers identify and evaluate vulnerabilities; then, based on how critical the vulnerabilities are, they release new updates. Manufacturers are required to transfer and install these updates on the devices securely. However, performing secure updates over the air is a challenging task since such smart devices have limited resources such as memory, battery, etc. Enabling secure updates in smart devices over the air is the primary aspect of the content of this thesis.

The secure protocols used by powerful computers around the world as a common language to talk to each other are unfit to address the limitations of smart devices. Hence, many other secure and lightweight protocols were designed for such devices. Manufacturers choose the best-suited protocol based on their devices' applications and requirements. Sometimes some devices need to talk to other devices or systems that use other protocols. There are tools such as protocol translators that can enable such communications. Enabling secure protocol translation is another aspect of this thesis.

The updates are usually targeting a large number of devices. Therefore, the devices can be grouped so that all group members can be updated at once instead of updating them one by one. In either group or one-to-one communication, the communication is secured through the use of secret keys. Secret keys are important in any secure communication, including applying updates. These secret keys must be distributed among the devices before the communication starts. Enabling secret key distribution for group or one-to-one communication is the final aspect covered in this thesis.

What we did?

Identifying and analyzing vulnerabilities. We designed a tool that can help manufacturers to identify, evaluate, and prioritize vulnerabilities. This tool can benefit manufacturers in the maintenance procedure of their products. We showed that our tool can be important to increase devices' security using the results obtained from participating manufacturers in our evaluation.
Analyzing existing common protocols for smart devices. We analyzed and compared two common protocols (CoAP and MQTT). We measured the energy consumption of a sample device in performing a secure update procedure over the air. Our evaluation indicated that each protocol has different merits depending on the specific application or use case.
Designing a new protocol for smart devices' updates. We designed a new secure update protocol for smart devices. We showed that the protocol consumes reasonable energy during the update procedure. It is also resistant to attacks such as attacks trying to consume the resources of smart devices. Therefore, the new protocol is an attractive choice for large-scale updates of devices with limited resources.
Designing a new architecture for protocol translation. We designed a protocol translator architecture that can securely convert one protocol to another. This architecture can be used by manufacturers designing products that use different protocols, and still, the devices are able to talk to each other. Using our architecture, the translation is done in an environment isolated from potential attackers. We showed that using such an isolated environment slightly increases the transmission time of the traffic. However, for critical security applications, this increase is acceptable given the increased security level which was achieved.
Designing new key establishment methods for smart devices. We designed one-to-one and group key establishment methods to share secret keys among smart devices. Manufacturers can use any of both methods based on their requirements and specifications. Our results indicated that both schemes are efficient and can be used in practice.