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

Nikbakht Bideh, Pegah; Gehrmann, Christian

Published in:
Proceedings of the 4th Workshop on CPS and IoT Security and Privacy

DOI:
10.1145/3560826.3563381

2022

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

Pegah Nikbakht Bideh
Christian Gehrmann
pegah.nikbakht_bideh@eit.lth.se
christian.gehrmann@eit.lth.se
Dept. of Electrical and Information Technology
Lund, Sweden

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.

CCS CONCEPTS

• **Security and privacy → Security protocols**.

KEYWORDS

Internet of Things, Over-The-Air, Secure code dissemination, Protected software upgrade

ACM Reference Format:
Pegah Nikbakht Bideh and Christian Gehrmann. 2022. RoSym: Robust Symmetric Key Based IoT Software Upgrade Over-the-Air. In *Proceedings of ACM Joint Workshop on CPS&IoT Security and Privacy (CPSIoTSec) (CPSIOTSEC 2022).* ACM, New York, NY, USA, Article 4, 12 pages. https://doi.org/xx.xxx/xxx_x

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 [13]. 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 [37] or through Denial of Service (DoS) of the actual update process [12]. 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) [7]. 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 [2]. 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 [5]. This is a problem when we consider resource constraint IoT units. On the other hand, to resist Grover's algorithm [16] 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 5, 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 [19]. 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 [11], 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 [27] 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 [8], 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 [20], 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 [17], SCATTER [26], SenSeOP [3] and Flexicast [28].

Similar to our solution, also different pure symmetric key based techniques have been introduced. PETRA [21] 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 [23], another symmetric scheme, similar to [8] 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 [30] 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 [40] 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 [24] 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 [24] indicates better performance compare to methods such as Seluge [20]. Hence, we find it relevant to compare our approach with SEDA as well and the figures are presented in Section 7. ASSURED [4] 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 [39] 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 [37] 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 [20] using a Merkle tree approach or the SECNRCC scheme [40], 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 [38], 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 [33]. 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., [12], 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 [33] 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 [12] 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 [12], 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 [37], 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 [22, 36], 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.

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 [33]. 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

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

Features	Cryptography	DoS resistance	Source authentication	Multicast
Deluge [19]	No security			
Secure Deluge [11]	Asymmetric		Digital signatures and packet level hash	
Sluice [27]	Asymmetric		Digital signatures and hash chains	
[8]	Asymmetric	Signed hash tree verification	Digital signatures	
Seluge [20]	Mixed	Immediate authentication, Message Specific Puzzle	Merkle hash tree and digital signatures	
SDRP [17]	Asymmetric		Merkle hash tree and digital signatures	
DiCode [18]	Asymmetric	Immediate authentication, Message Specific Puzzle	Merkle hash tree and digital signatures	
SCATTER [26]	Asymmetric	Signature verification	Merkle hash tree and r-time signatures	
SenSeOP [3]	Asymmetric	Signature verification and time lock	Hash value and digital signatures	Yes
SECNRCC[40]	Mixed	Weak authentication (delay)	Digital signatures	
Felxicast [28]	Asymmetric		Fingerprint with bloom filter	
[38]	Mixed	Encryption-Then-MAC approach, Cipher Puzzle	Digital signatures	
PETRA [21]	Symmetric		MAC with bloom-filter	
Castor[23]	Symmetric		MAC with hash chain	
μTESLA [30]	Symmetric		MAC with time synchronization	
SEDA [24]	Mixed	HMAC verification	HMAC and Digital signatures (Advertisement packets)	Yes
ASSURED [4]	Asymmetric		Digital signatures	
RoSym	Symmetric	Time limited MAC verification	MAC with hash chain	Yes

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 $I = I_0, I_1, I_2, \ldots, I_{n-1}$, i.e. the upgrade information is split into n distinct pieces.
- The software image is distributed using a sequence of software packages denoted by $P = P_0, P_1, P_2, \ldots, P_{n-1}$. 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 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 [20]. There are a number of broadcast weak authenticator mechanisms for symmetric encryption including *Message Specific Puzzle* (MSP) [18, 20] and *Cipher Puzzle* (CP) [38] 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 [1]. 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 [32, 35]. 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.

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

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 [14] 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 [31] or SHA-3 [10]. 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 [25] or short MAC [15].
(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.

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}}), $0 \leq i \leq n\text{-}2$
MAC_i	Message Authentication Code, MAC_i = MAC(IK_{sw}, {i, E(K_{sw}, I_i), h_{i+1}}), $0 \leq i \leq n\text{-}2$

Figure 1: RoSym software package format and size

5.3 Upgrade Procedure on IoT Unit Side

On the IoT side, the procedure starts with the actual firmware download credentials fetching and preparing as described in Section 5.1. Next, the IoT unit uses the supported firmware image fetch option in the system. 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 the actual software fetch 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 - $t_{i-1} > T_1$, $1 \leq i \leq n\text{-}1$, 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 - $t_c > T_2$, $0 \leq i \leq n\text{-}1$, 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 $0 \leq i \leq n\text{-}2$, and over the fields i and $E(K_{sw}, I_i)$, if $i == n\text{-}1$.
(6) Compare MAC_i with MAC'_i in the received package and only if coincide, accept the new package.
  - If $0 \leq i \leq n\text{-}2$:
    (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, $I = I_0, I_1, I_2,..., I_{n-1}$.

5.3.1 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.

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

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 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.

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.

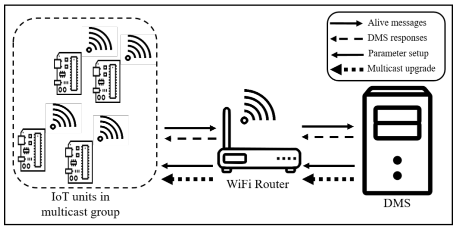


Figure 2: Information flow between DMS and IoT units

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 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.

[4]Our implementation is available at: https://github.com/pegahnikbakht/RoSym
[5]https://www.espressif.com/en/products/socs/esp32-s2
[6]https://github.com/espressif/esp-idf
[7]https://www.qoitech.com/

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 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).

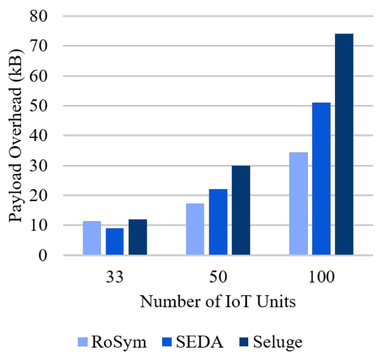
7.2 Communication Overhead

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 [24]). The efficiency of SEDA in a small sized network is due to the dependency of the key establishment on the neighboring units and not the whole network.

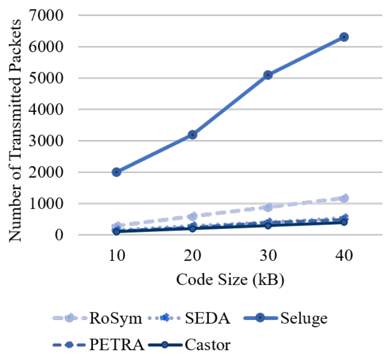
In Seluge the payload size is bound to 102 bytes and in order to have a fair comparison of the number of transmitted packets for different code sizes in the update phase, we bound the payload size of SEDA, PETRA, Castor, and our scheme to 102 bytes, although we can send larger payloads in our scheme. 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 [24]). 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 2. 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 2, RoSym almost doubles both energy



(a) Key establishment overhead



(b) Number of transmitted packets

Figure 3: Comparison of communication overhead in different schemes

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 2. 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 1200 μwh, 15.88 seconds with hardware accelerator, and 1170 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.

8 FORMAL SECURITY VERIFICATION

We formally model and verify the security properties of our designed solution using ProVerif [6] tool. ProVerif is an automated tool that is used in verifying the security properties of protocols and it uses Dolev-Yao model [9] as the adversarial model. In ProVerif applied pi calculus [34] 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.

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

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

8.2.1 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}.

8.2.2 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.

8.2.3 DoS Resistance. According to the Denial of Service definition which is defined in [29], 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 8.2.2 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.

ACKNOWLEDGMENTS

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

[1] Farah Afianti, Titiek Suryani, et al. 2018. Dynamic cipher puzzle for efficient broadcast authentication in wireless sensor networks. *Sensors* 18, 11 (2018), 4021.

[2] Konstantinos Arakadakis, Pavlos Charalampidis, Antonis Makrogiannakis, and Alexandros Fragkiadakis. 2021. Firmware Over-the-Air Programming Techniques for IoT Networks - A Survey. *ACM Comput. Surv.* 54, 9, Article 178 (oct 2021), 36 pages. https://doi.org/10.1145/3472292

[3] Nils Aschenbruck, Jan Bauer, Jakob Bieling, Alexander Bothe, and Matthias Schwamborn. 2012. Selective and secure over-the-air programming for wireless sensor networks. In *21st International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 1–6.

[4] N Asokan, Thomas Nyman, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. 2018. ASSURED: Architecture for secure software update of realistic embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2290–2300.

[5] Ward Beullens. 2021. *The Design and Cryptanalysis of Post-Quantum Digital Signature Algorithms.* KU Leuven. https://www.esat.kuleuven.be/cosic/publications/thesis-417.pdf

[6] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. 2018. ProVerif 2.00: automatic cryptographic protocol verifier, user manual and tutorial. *Version from* (2018), 05–16.

[7] Stephen Brown and Cormac Sreenan. 2013. Software Updating in Wireless Sensor Networks: A Survey and Lacunae. *Journal of Sensor and Actuator Networks* 2, 4 (Nov 2013), 717–760. https://doi.org/10.3390/jsan2040717

[8] Jing Deng, Richard Han, and Shivakant Mishra. 2006. Secure code distribution in dynamically programmable wireless sensor networks. In *Proceedings of the 5th international conference on Information processing in sensor networks.* 292–300.

[9] D. Dolev and A. C. Yao. 1981. On the Security of Public Key Protocols. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science (SFCS '81).* IEEE Computer Society, Washington, DC, USA, 350–357. https://doi.org/10.1109/SFCS.1981.32

[10] FIPS PUB DRAFT. 2014. 202. SHA-3 Standard: Permutation-Based hash and extendable-output functions. *Information Technology Laboratory, National Institute of Standards and Technology. Recovered on May* (2014).

[11] Prabal K Dutta, Jonathan W Hui, David C Chu, and David E Culler. 2006. Securing the deluge network programming system. In *5th International Conference on Information Processing in Sensor Networks.* IEEE, 326–333.

[12] Yan-Hong Fan, Mei-Qin Wang, Yan-Bin Li, Kai Hu, and Muzhou Li. 2021. A Secure IoT Firmware Update Scheme Against SCPA and DoS Attacks. *J. Comput. Sci. Technol.* 36, 2 (2021), 419–433. https://doi.org/10.1007/s11390-020-9831-8

[13] J. Ferreira, J. N. Soares, R. Jardim-Goncalves, and C. Agostinho. 2017. Management of IoT Devices in a Physical Network. In *21st International Conference on Control Systems and Computer Science (CSCS).* 485–492.

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

[15] Christian Gehrmann, Marco Tiloca, and Rikard Höglund. 2015. 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, 274–282.

[16] Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (Philadelphia, Pennsylvania, USA) *(STOC '96).* Association for Computing Machinery, New York, NY, USA, 212–219. https://doi.org/10.1145/237814.237866

[17] Daojing He, Chun Chen, Sammy Chan, and Jiajun Bu. 2011. SDRP: A secure and distributed reprogramming protocol for wireless sensor networks. *IEEE Transactions on Industrial Electronics* 59, 11 (2011), 4155–4163.

[18] Daojing He, Chun Chen, Sammy Chan, and Jiajun Bu. 2012. DiCode: DoS-resistant and distributed code dissemination in wireless sensor networks. *IEEE Transactions on Wireless Communications* 11, 5 (2012), 1946–1956.

[19] Jonathan W Hui and David Culler. 2004. 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.* 81–94.

[20] Sangwon Hyun, Peng Ning, An Liu, and Wenliang Du. 2008. Seluge: Secure and dos-resistant code dissemination in wireless sensor networks. In *2008 International Conference on Information Processing in Sensor Networks.* IEEE, 445–456.

[21] Wassim Itani, Ayman Kayssi, and Ali Chehab. 2009. 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.* 37–43.

[22] Irene Joseph, Prasad B. Honnavalli, and B. R. Charanraj. 2022. Detection of DoS Attacks on Wi-Fi Networks Using IoT Sensors. In *Sustainable Advanced Computing*, Sagaya Aurelia, Somashekhar S. Hiremath, Karthikeyan Subramanian, and Saroj Kr. Biswas (Eds.). Springer Singapore, Singapore, 549–558.

[23] Donnie H Kim, Rajeev Gandhi, and Priya Narasimhan. 2007. Exploring symmetric cryptography for secure network reprogramming. In *27th International Conference on Distributed Computing Systems Workshops (ICDCSW'07).* IEEE, 17–17.

[24] Jun Young Kim, Wen Hu, Hossein Shafagh, and Sanjay Jha. 2016. Seda: Secure over-the-air code dissemination protocol for the internet of things. *IEEE Transactions on Dependable and Secure Computing* 15, 6 (2016), 1041–1054.

[25] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. 1997. HMAC: Keyed-hashing for message authentication.

[26] Ioannis Krontiris and Tassos Dimitriou. 2011. Scatter–secure code authentication for efficient reprogramming in wireless sensor networks. *International Journal of Sensor Networks* 10, 1-2 (2011), 14–24.

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

[28] JongHyup Lee, LeeHyung Kim, and Taekyoung Kwon. 2015. Flexicast: Energy-efficient software integrity checks to build secure industrial wireless active sensor networks. *IEEE Transactions on Industrial Informatics* 12, 1 (2015), 6–14.

[29] Bo Meng, Wei Wang, and Wei Chen. 2012. Verification of Resistance of Denial of Service Attacks in Extended Applied Pi Calculus with ProVerif. *J. Comput.* 7, 4 (2012), 890–899.

[30] Adrian Perrig, Robert Szewczyk, Justin Douglas Tygar, Victor Wen, and David E Culler. 2002. SPINS: Security protocols for sensor networks. *Wireless networks* 8, 5 (2002), 521–534.

[31] FIPS PUB. 2012. Secure hash standard (shs). *Fips pub* 180, 4 (2012).

[32] Tie Qiu, Xize Liu, Min Han, Huansheng Ning, and Dapeng Oliver Wu. 2017. A secure time synchronization protocol against fake timestamps for large-scale Internet of Things. *IEEE Internet of Things Journal* 4, 6 (2017), 1879–1889.

[33] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin O'Flynn. 2017. IoT Goes Nuclear: Creating a ZigBee Chain Reaction. In *IEEE Symposium on Security and Privacy (SP).* 195–212. https://doi.org/10.1109/SP.2017.14

[34] Mark D Ryan and Ben Smyth. 2011. Applied pi calculus. In *Formal Models and Techniques for Analyzing Security Protocols.* Ios Press, 112–142.

[35] David Sanchez. 2007. 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.* 105–112.

[36] Zakir Ahmad Sheikh and Yashwant Singh. 2021. 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*, Pradeep Kumar Singh, Sławomir T. Wierzchoń, Sudeep Tanwar, Maria Ganzha, and Joel J. P. C. Rodrigues (Eds.). Springer Singapore, Singapore, 67–80.

[37] Jaewoo Shim, Kyeonghwan Lim, Jaemin Jeong, Seong-je Cho, Minkyu Park, and Sangchul Han. 2017. A case study on vulnerability analysis and firmware

modification attack for a wearable fitness tracker. *IT Converg. Pract* 5, 4 (2017), 25–33.

[38] Hailun Tan, Diethelm Ostry, John Zic, and Sanjay Jha. 2013. A confidential and DoS-resistant multi-hop code dissemination protocol for wireless sensor networks. *Computers & security* 32 (2013), 36–55.

[39] Luca Verderame, Antonio Ruggia, and Alessio Merlo. 2021. PATRIOT: Anti-Repackaging for IoT Firmware. *CoRR* abs/2109.04337 (2021). arXiv:2109.04337 https://arxiv.org/abs/2109.04337

[40] Mande Xie, Urmila Bhanja, Guiyi Wei, Yun Ling, Mohammad Mehedi Hassan, and Atif Alamri. 2015. SecNRCC: a loss-tolerant secure network reprogramming with confidentiality consideration for wireless sensor networks. *Concurrency and Computation: Practice and Experience* 27, 10 (2015), 2668–2680.