Rescuing LoRaWAN 1.0

Gildas Avoine
INSA Rennes, France
IRISA UMR 6074, Rennes, France
Institut Universitaire de France, Paris, France
gildas.avoine@irisa.fr

Loïc Ferreira
Orange Labs, Applied Cryptography Group, Caen, France
IRISA UMR 6074, Rennes, France
loic.ferreira@orange.com

Abstract

LoRaWAN is a worldwide deployed IoT security protocol. We provide an extensive analysis of the current 1.0 version and show that the protocol suffers from several weaknesses allowing to perform attacks, including practical ones. These attacks lead to breaches in the network availability, data integrity, and data confidentiality.
Based on the inner weaknesses of the protocol, these attacks do not lean on potential implementation or hardware bugs. Likewise they do not entail a physical access to the targeted equipment and are independent from the means used to protect secret parameters.
Finally we propose practical recommendations aiming at thwarting the attacks, while at the same time being compliant with the specification, and keeping the interoperability between patched and unmodified equipments.

1 Introduction

In parallel with the coming up of the Internet of Things, several communication protocols have been proposed, which technical specifics differ depending on the intended use case. For instance the Bluetooth wireless protocol [8] allows only short distance communication (several meters).[1] Technologies such as ZigBee [47] or Z-Wave [46] afford medium range distance communication (roughly a hundred meters) and aim at reducing the energy needed by the nodes to set up and maintain a mesh network.

As for long range distance communication (several kilometers), proposals have been made, such as LoRa. LoRa, developed by Semtech company, aims to set up a Low-Power Wide-Area Network (LPWAN), based on a long range, low rate, wireless technology. It is somewhat similar to a cellular technology (2G/3G/4G mobile systems) but optimised for IoT/M2M. LoRa does not require a spectrum license since it uses free (but regulated) radio spectrum (*e.g.*, 863-870 MHz in Europe, 902-928 MHz in the USA, 779-787 MHz in China) [25]. A LoRa device, with an autonomous power-supply, is supposed to communicate through several kilometers in an urban area, and to have a lifespan up to eight or ten years.
LoRaWAN is a protocol aiming at securing the Medium Access Control layer of a LoRa network. It is designed by the LoRa Alliance, an association gathering more than 400 members (telecom operators, semiconductor manufacturers, digital security companies, hardware manufacturers, network suppliers, *etc.*).

Public and private LoRaWAN networks are deployed in more than 50 countries worldwide [40] by telecom operators (SK Telecom, FastNet, ZTE, KPN, Orange, Proximus, *etc.*), private providers (*e.g.*, LORIOT.io [26]), and private initiatives (*e.g.*, The Things Network [44]). Several nationwide networks are already deployed in Europe (France, Netherlands) [14], Asia (South Korea) [27], Africa (South Africa) [6], Oceania (New Zealand) [41], providing coverage to at least half of the population. Trials are launched in Japan [9], the USA (starting with a hundred cities) [22], China (the expected coverage extend to 100 million homes and 300 million people) [39], India (the first phase network aims to cover 400 million people across the country) [21]. See Figure 1 for a worldwide map of LoRa networks.

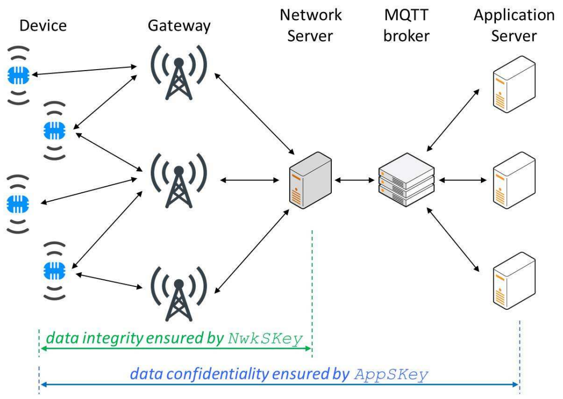
The version 1.0.1 followed by version 1.0.2 of the LoRaWAN specification has been released in 2016. In this paper we focus on this last version which is the released 1.0 version currently worldwide deployed.

1.1 Protocol overview

The LoRaWAN network corresponds to a star-of-stars topology: a set of devices communicates with several gateways which relay the data to a Network Server (NS) in the backend. In turn the NS delivers the data to

[1]That limitation is claimed in order to forbid data eavesdropping on the air interface.

Figure 1: Worlwide map of LoRa networks (source: [37])


Device
Gateway
Network Server
MQTT broker
Application Server
data integrity ensured by NwkSKey
data confidentiality ensured by AppSKey
Figure 2: LoRaWAN network (simplified view)

one or more Application Servers (AS) which own the corresponding device, optionally through intermediary servers such an MQTT server (see Figure 2). The security mechanisms are based on a symmetric key (the root key) `AppKey` shared between a device and the NS. From this key, distinct per device, two session keys are computed: the application session key `AppSKey` guarantees the data confidentiality between the device and the AS; the network session key `NwkSKey` guarantees the data integrity between the device and the NS (thus data integrity is not end-to-end provided between the device and the AS[2]). When a frame is exchanged exclusively between a device and the NS, both data confidentiality and data integrity are provided by the network session key `NwkSKey`. An application payload, if present, is always encrypted. If no payload is carried the frame is only authenticated. Encryption is done with AES [32] in CTR mode [12, 33], and data integrity is provided with AES in CMAC mode [35, 42]. A device may establish an "activation" (namely a session) with the NS through two ways. The pre-personalization (Activation by Personalization, ABP) consists in setting two session keys (and other parameters but not the `AppKey` root key) into the device before its deployment. An ABP device is then able to communicate with the NS (and its AS) but not to renew the "session" keys. The other possibility (Over the Air Activation, OTAA) consists in provisioning the device with an `AppKey` root key and other parameters, allowing to perform key exchanges with the NS through the radio interface once it is deployed.[3]

1.2 Contribution

Our contribution is twofold. Firstly we provide an extensive analysis of the protocol and show it suffers from several weaknesses. Then we describe how attacks, not only theoretical but also practical, based on the protocol flaws may be performed. Secondly we provide several recommendations aiming at mitigating the attacks while at the same time keeping the interoperability between a patched equipment and a non modified one. Our results show that all the attacks we describe may be thwarted if the recommended corrections are applied to the NS and the devices.

We emphasise that the aforementioned attacks, due to the protocol weaknesses, do not lean on potential implementation or hardware bugs, and are likely to be successful against any equipment implementing LoRaWAN 1.0. Likewise the attacks do not entail a physical access to the targeted equipment and are independent from the means used to protect secret values (*e.g.*, using a tamper resistant module such as a Secure Element).
Thus our attacker, standing between a LoRaWAN device and the NS, needs only to act on the air interface: she needs to eavesdrop on data exchanged between the device and the server, and to send data to these equipment. In particular the attacker do not need to get a physical access to the targeted device (or server).

We think that the countermeasures we propose represent straightforward changes to be implemented. Moreover the attacks we describe may allow to appreciate the security properties provided by the upcoming LoRaWAN version 1.1.
Some assessments we make, based on an independent study, are similar to other analyses [24]. However we present new attacks and for each of them we provide a precise description of its goal, its implementation, the technical means used, and the tangible consequences. Moreover our attacks do not lean on strong assumptions such as the ability to get a physical access to, and to monitor a device or the NS. In addition we describe attacks targeting either a device or the NS.

[2]As acknowledged by the specification ([43], §6.1.4).
[3]In this paper we focus on OTAA devices.

The attacks and their precise description, the adversary model (which does not imply a physical access to any equipment, in particular the device), and the two kinds of targeted equipment (device and server) are the main aspects that differentiate our work compared to previous works.

1.3 Paper outline

The LoRaWAN protocol is detailed in Section 2. Theoretical and practical attacks against LoRaWAN are described in Section 3 and Section 4. In Section 5 recommendations aiming at thwarting the attacks we describe are listed. Section 6 summarises previous comments and analysis on the protocol. And we conclude in Section 7.

2 The LoRaWAN protocol

The description provided in this section is based on the LoRaWAN 1.0 specification [43].

2.1 Key exchange

The key exchange done over the air is triggered when the device sends a Join Request message which the NS responds with a Join Accept message to. The (unencrypted) Join Request message includes two static IEEE EUI-64 identifiers (the device's `DevEUI`, and the AS' `AppEUI`), and a pseudo-random value `DevNonce` generated by the device. The message is protected with a 4-byte CMAC authentication tag (called `MIC`) computed with the 128-bit (static) root key `AppKey`. The Join Accept response from the NS contains the (static) identifier of the latter (`NetID`), a pseudo-random value generated by the NS (`AppNonce`), a value used as the device short address (`DevAddr`), and several (optional) radio parameters. The Join Accept message is protected with a CMAC authentication tag, and encrypted with AES (both operations made with the root key `AppKey`).[4] Two 128-bit session keys are then computed:

$$
\begin{aligned}
\texttt{NwkSKey} &= \mathsf{AES}(\texttt{AppKey}, \texttt{0x01}\|\texttt{data})\\
\texttt{AppSKey} &= \mathsf{AES}(\texttt{AppKey}, \texttt{0x02}\|\texttt{data})
\end{aligned}
$$

with

$$
\begin{aligned}
\texttt{data} = \texttt{AppNonce}\ (3)\|\texttt{NetID}\ (3)\|\texttt{DevNonce}\ (2)\\
\|\texttt{0x00}\cdots\texttt{00}\ (7)
\end{aligned}
$$

Thus the session keys depend mostly on a secret and static value (the root key `AppKey`), and two pseudo-random values of 2 and 3 bytes. Once the Join Request and Join Accept messages are exchanged, the device, the NS and the AS are able to communicate. After the NS computes the session keys, it transmits the application session key `AppSKey` to the AS, which has thus no control on this key sharing phase, entirely handled by the NS.[5] The NS must keep the previous session keys, and the corresponding security parameters, until it receives a (valid) frame protected by the new security parameters. The security mechanisms between NS and AS are out of the LoRaWAN scope. Figure 3 depicts an activation.

2.2 Data encryption and authentication

The frame payload `FRMPayload` is encrypted in CTR mode. From block counters

$$
\begin{aligned}
A_i = \texttt{0x01}\ (1)\|\texttt{0x00}\cdots\texttt{00}\ (4)\|\texttt{dir}\ (1)\|\texttt{DevAddr}\ (4)\\
\|\texttt{cnt}\ (4)\|\texttt{0x00}\ (1)\|i\ (1)
\end{aligned}
$$

a secret keystream $S_i = \mathsf{AES}(K, A_i)$, with $K \in \{\texttt{AppSKey}, \texttt{NwkSKey}\}$, is produced and used to mask the payload:

$$
[\texttt{FRMPayload}] = (S_0\|\cdots\|S_{n-1}) \oplus \texttt{FRMPayload}
$$

`dir` specifies the direction (uplink = `0x00`, downlink = `0x01`). `cnt` is the frame counter (of 16 or 32 bits), initialised to 0 when the session starts, and monotonically increased when a (valid) frame is sent or received. Two different counters are used depending on the frame's direction. `DevAddr` is the device address (within a given LoRa network) chosen by the NS and sent in the Join Accept message, and it remains constant during the entire session. To compute `DevAddr`, seven bits are chosen from the NS' unique identifier `NetID`: $\mathsf{msb}_7(\texttt{DevAddr}) = \mathsf{lsb}_7(\texttt{NetID})$, and 25 bits are "*arbitrarily*" assigned by the NS. The i value numbers the AES blocks within the payload to encrypt.

A 4-byte authentication tag is computed with CMAC and the network session key `NwkSKey` on the whole frame (header `hdr` of size $hlen \in \{8, \ldots, 24\}$ and encrypted payload [`FRMPayload`] of size $plen$) and a 16-byte prefix block

$$
\begin{aligned}
B_0 = \texttt{0x49}\ (1)\|\texttt{0x00}\cdots\texttt{00}\ (4)\|\texttt{dir}\ (1)\|\texttt{DevAddr}\ (4)\\
\|\texttt{cnt}\ (4)\|\texttt{0x00}\ (1)\|(hlen + plen)\ (1)
\end{aligned}
$$

Note that B_0 and A_i differ only on the first and last bytes, and share the same parameters `DevAddr` and `cnt`. The

[4]More precisely the AES decryption function is used to protect the Join Accept message, since the device implements the encryption function only.

[5]Note that, if the NS computes the session keys, it knows the application session key `AppSKey`, which is neither necessary, nor desirable. In practice, a third party may own the device's root key `AppKey`, derive the session keys `AppSKey`, and `NwkSKey`, and securely transmit the former to the AS, and the latter to the NS. This third party though is not specified in the LoRaWAN protocol 1.0, nor in any companion specification.

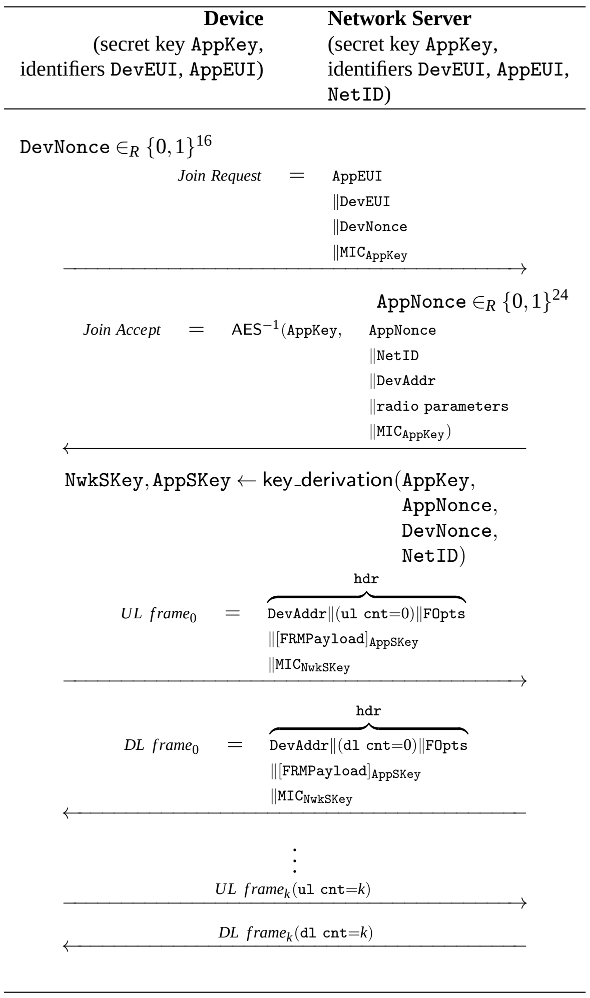

Device
(secret key AppKey, identifiers DevEUI, AppEUI)
Network Server
(secret key AppKey, identifiers DevEUI, AppEUI, NetID)
DevNonce ∈_R {0,1}^16
Join Request = AppEUI ||DevEUI ||DevNonce ||MIC_AppKey
AppNonce ∈_R {0,1}^24
Join Accept = AES^-1(AppKey, AppNonce ||NetID ||DevAddr ||radio parameters ||MIC_AppKey)
NwkSKey, AppSKey ← key_derivation(AppKey, AppNonce, DevNonce, NetID)
hdr
UL frame_0 = DevAddr||(ul cnt=0)||FOpts ||[FRMPayload]_AppSKey ||MIC_NwkSKey
hdr
DL frame_0 = DevAddr||(dl cnt=0)||FOpts ||[FRMPayload]_AppSKey ||MIC_NwkSKey
UL frame_k(ul cnt=k)
DL frame_k(dl cnt=k)

Figure 3: LoRaWAN activation (simplified scheme). Note that the frame direction (uplink, downlink) is involved in the payload encryption and the authentication tag computation.

frame eventually sent is

$$\texttt{hdr}\ (hlen) \| [\texttt{FRMPayload}]\ (plen) \| \texttt{MIC}\ (4)$$

Figure 4 depicts the generation of an application frame. The frame header `hdr` includes, among other fields, `DevAddr`, the frame counter `cnt` on 2 bytes[6], and an (optional) field `FOpts` which may contain commands[7] exclusively exchanged between the device and the NS.[8]

[6]If the frame counter is 32-bit long, this field corresponds to the least 16 significant bits.

[7]In the `FOpts` field these commands are in clear. If they have to be encrypted they must be included in the frame payload. In such a case the payload cannot contain application data, and the encryption key used is the network session key `NwkSKey`.

[8]The header of an uplink frame is not (completely) transmitted to the AS, as well as the `MIC` authentication tag.

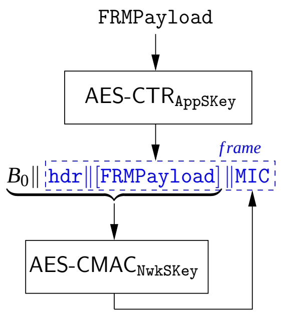


Figure 4: Generation of an application frame. Note that the session key `NwkSKey` may also be used to encrypt the `FRMPayload` payload.

2.3 Remark on encryption and authentication

At first glance the integrity and confidentiality mechanisms used in LoRaWAN follow the Encrypt-then-MAC paradigm, which generic security is proved by Bellare and Namprempre [5].[9] As for encryption the LoRaWAN specification explicitly refers to CCM* [16], which derives from CCM [34]. Regarding the authentication, LoRaWAN uses CMAC and not CBC-MAC as in CCM*. Note that LoRaWAN does not refer to CCM* as for data integrity but a prefix (B_0 block) is used in the computation of the LoRaWAN authentication tag, which format seems to be based on the B_0 prefix used in CCM*. The rationale behind this choice is unclear. CBC-MAC does not handle variable length inputs, and a way to tackle this restriction is to prepend the input length to the input [4] (CCM and CCM* follow that method). However LoRaWAN uses CMAC which allows arbitrary length inputs, and not CBC-MAC.

3 Attacks against LoRaWAN

Hereinafter we present our findings regarding the LoRaWAN protocol version 1.0, the currently deployed version. Table 1 summarises the attacks we have found against LoRaWAN.
We recall that our attacker stands between a device and the NS, and needs only to act on the air interface (to eavesdrop on data exchanged between the device and the server, and to send data to these equipment).

3.1 Replay or decrypt

In LoRaWAN encryption is done in CTR mode [12, 33] which security is proved by Bellare, Desai, Jokipii, and

[9]We do not claim that these security mechanisms *as provided* by LoRaWAN are secure, as we will see below.

Table 1: Attacks against LoRaWAN (n: number of DevNonce values the NS keeps track of. m: number of new session keys sets stored by the NS. D: device)

Attack	Cost (# Join message)	Probability of success	Impact
(A1) Replay or decrypt (D, §3.1.1)	$\simeq 2^{16}$	1	downlink frame replay, uplink frame decryption
(A2) Replay or decrypt method 1 (NS, §3.1.2)	1	2^{-24}	uplink frame replay, downlink frame decryption
(A2) Replay or decrypt method 2 (NS, §3.1.2)	$\simeq (n+1)\times 2^{24}$	1	uplink frame replay, downlink frame decryption
(A3) DoS (D, §3.2.1)	1	1	device disconnection
(A4) DoS (NS, §3.2.2)	m	1	device disconnection

Rogaway [3]. Likewise CMAC mode (namely OMAC1 [18, 19]), used to compute a frame's authentication tag, is proved secure by Iwata and Kurosawa [20], and Nandi [31]. Of course this does not necessarily imply that a protocol based on these cryptographic primitives is secure in turn [2, 11]. In particular the security of these encryption and authentication modes is no longer guaranteed in case of a misuse, namely if same session keys, counter blocks, and B_0 prefix block are reused. Based on the peculiarities of LoRaWAN, it is actually possible to compel a device or the NS to reuse previous security parameters. We describe precisely how to perform such an attack against a device or the NS, and its consequences.

3.1.1 Targeting a device

Goal The purpose of this attack is to compel the device to reuse previous session keys and other security parameters. When this happens, frames picked from a previous session become cryptographically valid anew, hence can be replayed. Moreover the same secret keystream is then used to protect the frames exchanged during the new session. This allows to attempt frame decryption.

Core The encryption keystream $S_i = \mathsf{AES}(K, A_i)$ used to protect a frame payload is produced from a session key $K \in \{\texttt{AppSKey}, \texttt{NwkSKey}\}$ and A_i block counters. Within a given session the blocks

$$A_i = \texttt{0x01}\ (1) \| \texttt{0x00}\cdots\texttt{00}\ (4) \| \texttt{dir}\ (1) \| \texttt{DevAddr}\ (4) \\ \| \texttt{cnt}\ (4) \| \texttt{0x00}\ (1) \| i\ (1)$$

(as well as the prefix block B_0) depend mostly on the frame counter cnt (set to 0 when the session starts and monotonically increased frame after frame), and on the DevAddr parameter (static during the whole session). The other parameters are the direction dir unchanged for a given direction, and the i block index which evolves the same way for each frame. Hence the way the keystream S_i changes depends only on the DevAddr parameter and the session key (usually AppSKey). For a given device, which connects to the same NS (hence uses the same static NetID parameter), the session keys depend mainly on a secret and static value (AppKey) and two pseudo-random values (DevNonce, AppNonce). Therefore, if one succeeds in compelling the device to reuse the same DevAddr, DevNonce and AppNonce parameters, this leads not only to the reuse of previous session keys AppSKey, and NwkSKey, but also to the reuse of previous keystream S_i and prefix block B_0.

Attack The purpose is to make the device use twice the same DevNonce, AppNonce, and DevAddr values. The 2-byte DevNonce and 3-byte AppNonce parameters are pseudo-random. Hence two such values repeat with high probability ($p = \frac{1}{2}$) after roughly $\sqrt{2\ln(2)\times 2^{40}} \simeq 1.23\times 10^6$ activations done between the targeted device and the NS, due to the birthday paradox. If a device performs one activation per day, this corresponds to more than 3382 years. Let us assume that an attacker is able to impose the AppNonce value the device uses to compute the session keys. Then the probability that the session keys repeat depends only on the DevNonce parameter. In such a case, the collision happens with high probability after roughly $\sqrt{2\ln(2)\times 2^{16}} \simeq 301$ activations only. Even if the device does one activation per day, the reuse becomes possible after 10 months. Waiting for such a collision is a way to perform an "opportunist" (and very long) attack. In such a setting the attacker just waits for the device to launch 301 different sessions, and passively eavesdrops on the frames exchanged with the NS. Then the attacker gets (with high probability) two different sessions protected with the same security parameters.

Another option is to speed up the whole process. First the attacker eavesdrops on a given session, and she compels the device to generate multiple DevNonce values until the expected value is produced once again. In such a case only one value among 2^{16} is useful to the attacker, hence the device must generate on average 2^{16} DevNonce values.

The shortest receiving window of a Join Accept message is 5 seconds [25]. Assuming that the time needed to process the Join messages is negligible compared to the communication duration, the attack is achieved after roughly 91 hours.
A self-powered LoRaWAN device is expected to have a lifespan up to ten years. The computations made to produce an application frame or a Join Request message, or when receiving a Join Accept message are not the

same. However the energy cost to transmit and to receive data usually exceeds the cost of cryptographic processing, hence we may neglect the latter [38]. If the device sends one message per hour, including one Join Request per day, and receives one Join Accept per day, it should be able to handle $25 \times 365 \times 10 = 91{,}250$ messages during its lifetime. The attack requires the device to handle two Join messages per try, that is on average $2 \times 2^{16} = 131{,}072$ messages. If the device is self-powered the attack may run through its battery turning it into a denial of service ending in the permanent device deactivation. On the other hand the device may be connected to an external power source.

Once this first phase of the attack is achieved, the attacker ends with two different sessions protected with the same security parameters. Let s_{new} be the (new) session during which the same security parameters are used than during a previous session (s_{old}).

Technique 1 used to achieve the attack: replay of a Join Accept message In order to compel the device to use a given `AppNonce` value, the attacker can replay a previous Join Accept message to the device. Then the device will reuse (once again) the parameters included in the message. Indeed the data contained in a Join Accept message correspond to

$$\texttt{AppNonce}\ (3) \| \texttt{NetID}\ (3) \| \texttt{DevAddr}\ (4) \\ \| \texttt{radio parameters}\ (2 \ldots 18) \| \texttt{MIC}\ (4)$$

where `MIC` is an authentication tag computed on the preceding fields with the (static) root key `AppKey`. These parameters are protected with AES and `AppKey`. Note that all the parameters are chosen by the NS, in particular `AppNonce` and `DevAddr`. `NetID` is the NS' (static) identifier, and the `radio parameters` are also defined by the NS. The only secret parameter involved in the message calculation is static (`AppKey`). Hence the device is not able to verify if the received Join Accept message corresponds to the Join Request it sent. Replaying a Join Accept message allows the attacker to compel the device to (re)use both `AppNonce` and `DevAddr` parameters.

The possible choices for the attacker bear on the Join Accept messages previsouly sent by the NS to the targeted device. A Join Accept message intended to another device is not usable since the message is protected with the device's root key.

Technique 2 used to achieve the attack: harvest of Join messages The ability of the attacker to make the device generate multiple `DevNonce` values is related to the issue regarding the expected behaviour of the device when it sends a Join Request message but does not receive a Join Accept response or receives an invalid message. The specification is unclear about the following alternative: send again the same Join Request message (with the same `DevNonce` value), or generate a new `DevNonce` value and create a fresh Join Request message. However, it states that the NS shall ignore Join Request messages containing previously used `DevNonce` values in order to thwart a replay attack ([43], §6.2.4). Hence we may assume that the device generates a new pseudo-random `DevNonce` value each time it computes a Join Request message, even when a previous Join Request message did not receive a response. Otherwise the device may fear the subsequent Join Request messages to be dropped by the NS. This allows the attacker to collect multiple new and valid Join Request messages. It is enough for the attacker to send "false" Join Accept messages in response to the device's messages. Moreover, if the attacker forbids the NS from receiving the Join Request messages sent by the device, he gets "fresh" messages (*i.e.*, unknown to the NS) for free. In order to make the device start producing the Join Request messages, the attacker may wait or force (once only) the device to start a new session (*e.g.*, the attacker may turn the device off and on; once the power supply is re-established, the device likely starts a new activation).[10]

Note that every time the NS receives a Join Request message, it sends a new Join Accept message. Hence, this procedure is also a way to collect multiples Join Accept messages (when the attacker does not forbid the NS from receiving each Join Request message).

Impact: frame replay Frames drawn from the previous session (s_{old}) can be replayed to the device throughout the new session (s_{new}).[11] These frames are valid since they are protected with a cryptographically correct keystream and authentication tag. The attacker has to take care about the sequentiality. Indeed a frame shall be rejected by the device if its counter does not belong to $\{cnt, \ldots, \texttt{MAX\_FCNT\_GAP}\}$, where cnt is the dowlink frame counter (in that specific case) managed locally by the device and used as reference (its initial value is 0), and $\texttt{MAX\_FCNT\_GAP} = 2^{14}$. Hence the attacker may virtually choose up to $2^{14} + 1$ frames in order to deceive the device (more precisely the number of available frames depends on the number of frames actually sent by the NS

[10]Being able to influence on the power supply does not necessary mean to have a physical access to the device. The attacker could turn off or interrupt a remote electric generator the device is connected to, or the link between the generator and the device (if the device is powered by an external source), or use other means (*e.g.*, electromagnetic impulse targeting the device and leading to a power outage).

[11]We use the term "session", yet it is a misuse of language. Indeed this word does not depict precisely what are the actual exchanges since the device, at this point, has no "partner": neither the NS nor the AS is able to communicate with the device, and the attacker is unable to forge new valid frames.

or the AS throughout session s_{old}). Note that the first frame the attacker replays may be any of these. However the subsequent replayed frames must have increasing counter values.

Impact: frame decryption The frame payload is encrypted in CTR mode. Once the attack is achieved, the device uses twice the same keystream in order to protect different frames. The frame of counter t sent during session s_{old} contains an encrypted payload $c_t^{S_{old}} = m \oplus k_t^{S_{old}}$, where m is the clear data and $k_t^{S_{old}}$ the keystream. The frame of same counter t sent during session s_{new} contains an encrypted payload $c_t^{S_{new}} = m' \oplus k_t^{S_{new}}$. Since $k_t^{S_{old}} = k_t^{S_{new}}$, we have that $c_t^{S_{old}} \oplus c_t^{S_{new}} = (m \oplus k_t^{S_{old}}) \oplus (m' \oplus k_t^{S_{new}}) = m \oplus m'$. Hence m and m' may (partially or completely) be retrieved (in an obvious manner if one message, m or m', is known, or through analysis of $m \oplus m'$ [28]).

According to the LoRaWAN specification ([43], §4.3.1.1), if the device sends more than ADR_ACK_LIMIT $= 64$ frames to the NS, it has to ask an explicit acknowledgment to the server (the device sets the ADRACKReq bit to 1 within the frame header). The device can then send up to ADR_ACK_DELAY $= 32$ more frames to the server. If still no frame has been received from the server, the device *may* switch to the next lower data rate that provides a longer radio range. Furthermore, if the device already uses its lowest available data rate, it *shall* not ask for such an acknowledgment. The specification provides no guidance on how the device shall behave in the latter case, or if it still does not receive an acknowledgment after it changed its data rate, or if it decides not to change its rate. We may reasonably assume that the device keeps sending frames "normally". This means that the attacker has at her disposal at least ADR_ACK_LIMIT + ADR_ACK_DELAY $= 64 + 32 = 96$ frames usable for her decryption attempts. Moreover, if the device asks for an acknowledgment (the attacker is aware of that since the information ADRACKReq lies in the – unencrypted – frame header), the attacker can use any downlink frame drawn from session s_{old}, and replay it to the device. Indeed, according to the specification, this is enough to respond to the acknowledgment request sent by the device.

Cause This attack, allowing to compel the device to reuse the same security parameters (session keys, keystream, and prefix B_0), is possible because the DevNonce values are short and pseudo-random, hence may repeat "quickly", and the device has no means to detect if the AppNonce and DevAddr values repeat. Being able to detect such a replay would not necessarily allow the device to eventually compute shared session keys with the NS, but this would at least prevent the device from sending (new) frames protected with reused security parameters, hence avoiding the exploitability of the attack.

3.1.2 Targeting the NS

Goal The same kind of attack can be performed against the NS, aiming at compelling the server to use the same security parameters throughout two different sessions. The goal is then to compel the NS to use twice the same DevNonce, AppNonce and DevAddr values.

Attack: method 1 The key exchange is triggered by the Join Request message. Hence an attacker replaying a Join Request message sets the DevNonce value before knowing the DevAddr and AppNonce values the NS generates. These values must correspond to the DevNonce value chosen by the attacker, hence only one such couple among all possible values is of interest to the attacker.

According to the specification, the NS must keep track of "*a certain number*" of received DevNonce values in order to prevent replay attacks, without clarifying if this means all values or a few of them. We may reasonably assume that the NS keeps track of a few values (say n). Thus the attacker cannot choose any Join Request she wants to replay. The corresponding DevNonce value must not belong to the list of n stored values. If the value the attacker wants to replay still belongs to the server's list (let i be its index, with 0 and $n-1$ the index of the oldest and of the latest received values), he has to wait for $i+1$ additional (legitimate) key exchanges before the NS "forgets" that value. The duration of such an "opportunist" attack depends on the frequency of the key exchanges.

The purpose of the attacker is to get the same two AppNonce and DevAddr values as during a previous session. AppNonce is a 3-byte pseudo-random value. The 32-bit DevAddr parameter is made of 7 bits from NetID, and 25 bits which are "*arbitrarily*" chosen by the NS ([43], §6.1.1). If DevAddr is pseudo-random then the probability of success is $2^{-(24+25)} = 2^{-49}$. But "*arbitrarily*" does not mean "pseudo-random" and experiments we have made shows that the DevAddr parameter remains unchanged for a given device throughout different sessions.[12] In such a case the probability of success increases to 2^{-24}, and the overall probability of success is 2^{-24} every $n+1$ sessions. Note that the attacker can send in parallel several Join Request messages (each supposedly coming from a different device), hence increasing

[12]Thus some NS implementation derives the DevAddr parameter from the unique device's identifier DevEUI. Also the DevAddr value may be chosen once and for all at the time of the device provisioning.

the overall probability of success by the number of messages.

The attacker is successful when the NS sends the expected AppNonce value. But, contrary to a Join Request message (in clear), a Join Accept message is protected with AES. Before encryption a Join Accept message corresponds to

$$\texttt{AppNonce}\ (3) \| \texttt{NetID}\ (3) \| \texttt{DevAddr}\ (4) \\ \| \texttt{radio parameters}\ (2 \ldots 18) \| \texttt{MIC}\ (4)$$

NetID is the unique NS' identifier, hence static. As observed experimentally, the DevAddr parameter assigned to a given device remains unchanged. The radio parameters (frequency plan) depend on the gateway, hence likely remain the same for quite a long time. And MIC is an authentication tag computed on the preceding values with the device's (static) root key. Hence only AppNonce may vary from one Join Accept message to another. Through direct comparison between a Join Accept message (received during the attack) and the one used as reference (beforehand eavesdropped by the attacker during session s_{old}), the attacker is able to check if the AppNonce value repeats.

Attack: method 2 Alternatively the attacker can do the following. In a first phase the attacker collects a set of $n+k$, $k \geq 1$, different Join Request messages from a given device (hence the list exceeds the NS "memory"). Necessarily at least one of these messages contains a DevNonce value which is "forgotten" by the NS. The other messages may carry DevNonce values known at the moment to the NS. These messages must be ordered the following way: first the "forgotten" messages, followed by the others sorted in the same relative order than in the NS' list.[13] In a second phase the attacker sends continuously each Join Request message of its circular list. Hence the attacker has to make on average $(n+k) \times 2^{24}$ tries before getting one of the expected AppNonce values. For instance, with $n = 10$, the duration of the attack is higher than 29 years (with a key exchange done in 5 seconds).

Means used to achieve the attack In order to collect the Join Request messages, the attacker can iterate $n+k$ times the procedure described in Section 3.1.1 and targeting a device. Then the attacker gets a list of messages correctly sorted and ready to be used. The Join Request messages usable by the attacker must come from the same device since the session keys are computed with the device's root key (and also if the DevAddr parameter is closely related to the device – *e.g.*, computed from its DevEUI identifier).

[13]More exactly the necessary condition is the following: each message collected by the attacker and common with the NS' list must have a greater index than the index in the server's list (0 being the index of the oldest message, $n-1$ the index of the latest), so that the message is replayed once it has left the server's list.

Impact Once the attacker succeeds in compelling the NS to compute once again the same security parameters, she eventually gets two different sessions (s_{old} and s_{new}) protected with the same security parameters. The attacker is then able to replay uplink frames and attempt decryption of downlink frames.

According to the specification, when a new key exchange is done the NS shall keep the previous security parameters until it receives a (valid) frame protected with the new security parameters, and then it can remove the previous ones ([43], §6.2.4). Yet, if the NS has to send frames, likely it uses the latest security parameters. Yet, since the session keys and other security parameters are reused, the attacker can easily replay to the NS a frame drawn from the previous session s_{old}, thus "confirming" the new keys to the NS. Then the server drops the current keys and is ready to use the new ones.

Cause This attack, allowing to compel the NS to reuse the same security parameters (session keys, keystream, and prefix B_0), is possible because the AppNonce values are short and pseudo-random (hence may repeat), and the means used by the NS in order to detect a DevNonce reuse (storage of a – short – list of values) is not reliable.

3.2 Denial of service

3.2.1 Targeting a device

Goal This attack aims to "disconnect" the device from the network. That is the device performs a successful key exchange which ends with the device not sharing the new session keys with the NS (the device has no "partner"). Therefore the frames sent by the device are ignored by the NS, and conversely.

Core The session keys are computed, by a given device and the NS, with two static parameters (the NS' unique identifier NetID, and the device's root key AppKey), and two variable parameters (the pseudo-random values AppNonce computed by the NS, and DevNonce by the device). As soon as the device receives a (valid) Join Accept message it can derive the session keys and start transmitting protected frames. If the device uses, in the key derivation, values different from those actually sent by the NS (say $(\texttt{DevNonce}, \texttt{AppNonce}) = (x, \tilde{y})$ on the one hand, and $(\texttt{DevNonce}, \texttt{AppNonce}) = (x, y)$, on the other hand, $y \neq \tilde{y}$), it eventually computes different session keys than those computed by the server. Yet this does not forbid the device to send protected frames. However those frames will be dropped by the NS (since they are invalid from the server perspective), and, conversely, the frames sent by the NS will be discarded by the device. Thus the device, unable to communicate with the NS, is "disconnected" from the network.

Attack In order to perform such a denial of service (DoS) attack, an attacker can first passively eavesdrop on a Join Accept message sent by the NS in response to the device's Join Request message. When the device starts a new session and sends another Join Request message, the attacker replies before the NS and replays the eavesdropped Join Accept message. Likely this message contains an $\texttt{AppNonce} = \tilde{y}$ value different from the one sent by the NS ($\texttt{AppNonce} = y$). Hence the device and the NS compute different session keys and security parameters.

Means used to achieve the attack The attacker is able to replay a previous Join Accept message thanks to the peculiarities of the LoRaWAN protocol: indeed the device has no means to verify neither if the message is a replay, nor if it is an actual response to the Join Request message it just sent. Moreover the attacker can use the procedure described in Section 3.1.1 to collect several Join Accept messages and use these "DoS ammunition" anytime later. The Join Accept message used by the attacker must be intended to the targeted device. Indeed such a message is protected with the root key of the device it is sent to.

Impact Such a DoS attack may be harmful because it can lastingly disturb the operating of a LoRaWAN network. So then the usual behaviour of a sensor may be to regularly send some measurements without expecting a response unless the server detects an anomaly in the collected data. If the device sends its measurement at a low rate, days or even weeks may elapse before something abnormal is noticed, even if the device is supposed to react if it does not receive a downlink frame after a fixed number of sent frames. For instance, if the device sends one frame per hour, at least four days ($\texttt{ADR\_ACK\_LIMIT} + \texttt{ADR\_ACK\_DELAY} = 64 + 32 = 96$ hours) may slip.

Cause This DoS attack allowing to disconnect the device from the network is possible because the device is not able to check if a received Join Accept message corresponds to the Join Request message it sent, and uses new session keys without verifying it actually shares the same keys with the NS [15].

3.2.2 Targeting the NS

Goal The same kind of DoS attack can be done against the NS, aiming at disconnecting a given device from the network. In that case, the NS completes the key exchange without being "partnered" with the intended device (*i.e.*, identified by the `DevEUI` parameter within the Join Request message). Therefore the frames the NS (or the AS) may send are ignored by the device, and conversely.

Attack As soon as the NS receives a (valid) Join Request message it generates a new `AppNonce` value and computes new session keys. If an attacker succeeds in replaying to the NS a valid Join Request message, the corresponding device will no longer share the same session keys with the NS.

If the attacker owns a Join Request message she can send it anytime to the NS. However if the attacker replays a previous Join Request message, that message may be rejected by the NS since it is supposed to keep track of previously received `DevNonce` values. This means that the attacker has to expect, or to wait, for the `DevNonce` value included in the replayed Join Request message to no longer belong to the server's list (*i.e.*, the attacker has to wait for the targeted device to start enough activations so that the server "forgets" that `DevNonce` value). Alternatively the attacker may send a brand new Join Request message to be sure that it is not rejected by the NS. However, the message is protected by a 32-bit authentication tag computed with the device's root key `AppKey`. Hence the probability for the attacker to forge such a valid message is 2^{-32}.

A trade-off is the following: the attacker uses a fresh Join Request message, and leans on the targeted device to compute such a message, while forbidding the NS from receiving it (at the moment of its collection). Therefore the message is at the same time valid and unknown to the NS.

The attacker has another challenge to take up. The specification states that the NS must keep the previous session keys (and the corresponding counters) until it receives a valid frame protected with the new keys, and then it can drop the previous security parameters and keep only the new ones.[14] Yet it is unclear about how the NS should behave if it receives successively several valid Join Request messages but no frames protected with any of the new computed session keys. We may assume that the NS stores at most a few number of security parameters. Let us assume that the NS stores only two sets of session keys: the latest valid one, and the latest

[14]Note that this is introduced in version 1.0.2 of the specification and does not appear in version 1.0.1. Also the LoRaWAN specification does not demand the same regarding the device.

computed one. Let $\texttt{seskey}_i$ be the current (valid) session keys (used by the device and the NS to exchange frames). The attacker can do the following. She waits for the device to start a new activation. New session keys ($\texttt{seskeys}_{i+1}$) are then computed. The device stores $\texttt{seskeys}_{i+1}$ only while the NS stores both $\texttt{seskeys}_i$ and $\texttt{seskeys}_{i+1}$. Before the device sends a frame, the attacker immediately sends to the NS a Join Request message she previously eavesdropped on (and not received, hence new to the server). The server computes new session keys $\texttt{seskeys}_{i+2}$ which replace the unconfirmed keys $\texttt{seskeys}_{i+1}$. Then the NS stores $\texttt{seskeys}_i$ and $\texttt{seskeys}_{i+2}$ while the device stores $\texttt{seskeys}_{i+1}$. Hence the device and the NS do not share the same session keys. More generally, if the NS keeps the latest valid session keys and m new sets of keys, the attacker must send successively m new Join Request messages in order to "desynchronise" the NS and the device.

This attack is based on the ability for the attacker to gather multiple and new Join Request messages (*i.e.*, fresh DevNonce values). However if the device sends again the same Join Request message when it does not receive a valid answer from the NS, then it is possible to disconnect the device once and for all. Indeed, in such a case, if the device does not receive a valid Join Accept message when it starts a new activation (*e.g.*, the attacker sends a "false" Join Accept message before the NS), it will keep sending the (same) request which is then continuously discarded by the NS since it has already received the message. Hence the device is stuck.

Means used to achieve the attack In order to get a new Join Request message the attacker can use the technique described in Section 3.1.1 aiming at compelling the device to generate multiple Join Request messages. The attacker can gather several such messages and use these anytime later as "DoS ammunition".

Impact The consequences of this attack against the NS are the same as the one against the device: the targeted device is disconnected from the network. Unaware that the NS does not share the same security parameters, it may keep sending uplink frames for quite a long time while the NS is unable to process them. Conversely, the frames the NS may send cannot be understood by the device.

Cause This DoS attack against the NS and targeting the device is possible because the means used by the NS to detect a replay of a Join Request message is not reliable, and the device uses new session keys without verifying it actually shares the same keys with the NS.

3.3 Extended attack surface

The threats and attacks described in the previous sections assume that each device owns a different root key AppKey, in accordance to what the specification demands. However we may not exclude that an application provider deploys the same root key on a set of devices it owns. In such a case, the attacks described above in sections 3.1 and 3.2 may be enhanced. We stress that the specification demands to use a distinct root key per device (see [43], §6.2.2, p. 33). Here the purpose is to enlighten about the consequences (also due to the specifics of the LoRaWAN protocol) of what can be seen by some application providers as a slight deviation from the specification.

3.3.1 Replay or decrypt

Goal If several devices share the same root key AppKey, an attacker is able to replay to some device B, a Join Accept message sent by the NS to a device A. In such a case, if both devices (connected to the same NS) use, in addition, the same DevNonce value, they necessarily compute the same session keys, keystream, and B_0 prefix (since they both received the same DevAddr and AppNonce parameters, and share the same NetID value). Note that, if the Join Accept message is accepted by both devices, the Join Request message sent by each of them differs (at least) on the device's identifier DevEUI. However this parameter is not involved in the security parameters computation.

Attack In order to force a device B to generate a given DevNonce value (similar to the one used by device A), the attacker may compel device B to generate on average 2^{16} Join Request messages until she gets the expected one (using the technique described in section 3.1.1). An alternate option is to lean on the number of devices deployed (with the same root key). If more than $\sqrt{2\ln(2)\times 2^{16}} \simeq 301$ devices are deployed, two of them are going to generate the same DevNonce value with high probability ($p = \frac{1}{2}$). As soon as the attacker detects such a collision between two devices A and B, he can replay to B the Join Accept message received by A.[15] Then the attacker can exploit the (legitimate) session between device A and the NS to attack device B (trying frame replay and frame decryption).

Conversely the attacker can exploit the "session" started by device B to target the NS. Since devices A and B then share the same DevAddr value, the NS is unable to distinguish frames sent by each device. Hence the first frames send by B, if received, are likely dropped by the

[15]Note that all devices transmit on a limited number of radio channels.

NS which considers that they come from A with an incorrect counter. The attacker has to expect, or to force, device B to send enough frames so that its uplink counter reaches A's one. Conversely if the NS receives from device B a frame with a higher counter than device A, some frames sent by A may be discarded by the server as they carry an invalid (lower) counter. [16]

3.3.2 Denial of service

Goal Sending, before the NS, a Join Accept message to a device in response to its Join Request allows to disconnect the device from the network.

Attack In this setting the Join Accept messages usable by an attacker are not only the previous messages sent by the NS to the targeted device, but also any such message sent by the NS to all devices (since they share the same root key `AppKey`). Hence the number of usable "DoS ammunition" is notably augmented.

4 Lack of data integrity

The LoRaWAN protocol aims to provide data confidentiality and data integrity on the air interface, between the device and the NS. However the data exchanged between the NS and the AS are only encrypted but not integrity protected since the NS is the only one to own the key used to compute an authentication tag. Therefore the AS is not able to verify if a (encrypted) payload has been modified. The specification recommends to implement (at the application level) an integrity protection mechanism if the application provider wishes to do so. Moreover the specification seems to imply that such a mechanism is in fact optional since "*Network servers are considered as trusted*" ([43], §6.1.4, p. 32). This is a bold statement. Firstly the threat may not come only from the NS (even if it can be dishonest or compromised). Indeed an attacker may target the link (and intermediary servers) between the NS and the AS. Secondly encryption only does obviously not provide data integrity, but it may even not be sufficient to guarantee data confidentiality (in particular in the LoRaWAN case).

Goal The purpose of the attacker is either to modify or to decrypt an encrypted payload. The frame carrying the payload may be sent by the device to the AS or sent in the converse direction. Contrary to the attacks described in Section 3, our attacker here is able to act on the link between the NS and the AS. For instance the attacker could target and try to intrude on a (not or poorly protected) MQTT server used to relay data between NS and AS [36].

Attack on data integrity Data encryption is done in counter mode, therefore it is possible to change the plaintext by flipping bits of the ciphertext. If the content of an encrypted payload or merely the format of the unencrypted content is known, the attacker can replace or alter the data with accuracy. For instance, if the device is a sensor the attacker could change the measurement (temperature, humidity, *etc.*) sent. If the device is a presence sensor, the attacker may change a (binary) value notifying an intrusion into the opposite value notifying that everything is quiet. If the device is an actuator, the attacker could change a command ordering to close a window into a command ordering to open it. The attacker could also truncate the encrypted payload in order to hide information to the AS or to the device.

If the recipient is the AS, it is not able to detect that the payload is modified since there is no authentication tag. If the recipient is the device, it is not able to detect the modification since the authentication tag is computed by the NS after the attacker modifies the frame (in fact the device will validate the frame).

Attack on data confidentiality The attacker may try to guess the plaintext corresponding to an encrypted frame as follows. The attacker eavesdrops on a frame which payload is of the form $c = k \oplus m$, where k is the keystream, and m the plaintext to recover. She makes a guess m' regarding the plaintext, and chooses a message u from a set of valid applicative messages (*e.g.*, a finished list of commands shared by the device and the AS). The attacker computes $c' = c \oplus (m' \oplus u)$, and sends c' (to the server or the device). If the guess is correct ($m = m'$) then $c' = c \oplus (u \oplus m') = c \oplus (u \oplus m) = k \oplus u$. Hence the decryption will be correct and the command will likely be completed. Using this kind of "command oracle" attack [1, 10, 17], the attacker may rely on the expected behaviour of the recipient (either the device or the AS) to understand if his guess is correct.[17]

If the recipient is the AS, the attacker can make several tries (using the same encrypted payload and frame counter), because unlikely the AS verifies the frame counter (since the NS does it). On the contrary, if the recipient is the device, likely the attacker can make one try only, because the device verifies the frame counter and will reject subsequent downlink frames carrying a reused counter.

[16]And the attacker must use another criterion than `DevAddr` to discriminate the frames sent by the two devices (*e.g.*, some characteristics of the radio signal, such as its intensity).

[17]If the attacker targets the device, he may do experiments with one such specimen he owns in order to learn first how the device behaves, before acting.

Attack on data authenticity If the attacker succeeds in recovering a keystream k it can forge any ciphertext of her choice. For instance the attacker can change a set of data $(\texttt{DevAddr}, \texttt{FCntUp}, c = k \oplus m)$ into $(\texttt{DevAddr}, \texttt{FCntUp}, c' = k \oplus m')$. Since the uplink counter is verified by the NS, likely the AS does not check it, and uses the received DevAddr and FCntUp values in order to decrypt c'. Conversely the attacker can change a set of data $(\texttt{DevAddr}, \texttt{FCntDown}, c = k \oplus m)$ laying on the MQTT server into $(\texttt{DevAddr}, \texttt{FCntDown}, c' = k \oplus m')$, hence deceives the device.

The attacker can recover the keystream if he knows the corresponding plaintext. If the attacker succeeds in decrypting data through the "command oracle" attack described above, he also gets the corresponding keystream (partially or totally). Then he can use it to forge encrypted payloads intended to the AS. However, it is unlikely that the device accepts the forged payload. Indeed in such a case, the device has already received the frame corresponding to some counter FCntDown, hence it will discard any other frame (the one forged by the attacker) carrying the same counter FCntDown.

This attack is not due to a lack of protection of some intermediary server (between the NS and the AS, such as an MQTT server). If the application frames were duly protected, the worst an attacker could do would be to delete frames. However, since LoRaWAN does not provide end-to-end integrity protection between the device and the AS, it is possible to deceive both of them. Therefore we strongly recommend to implement integrity protection between the device and the AS.

Moreover the AS should not blindly trust the NS and should verify every security parameter it receives from third parties. In particular, if the AS receives the application session key AppSKey from the NS, it should verify that the key is fresh (not reused). Similarly the AS must keep track of the frame counters (both uplink and downlink counters) in order to avoid frame replays.

5 Recommendations

In this section we aim at providing recommendations that thwart the attacks described in Section 3. This may lead to major changes in the protocol specification and break the interoperability between patched and non-modified equipment. Hence, as an additional constraint, we aim at proposing improvements that could solve the issues as best as possible while retaining at the same time the compliance with unchanged version of devices or servers, in particular equipment that are already deployed and may not be easily patched.

Table 2 summarises the proposed countermeasures.

5.1 List of the possible recommendations

5.1.1 Detect the replay of a value

The crux of the "replay or decrypt" attack is to compel the victim (device or NS) to use a parameter (AppNonce or DevNonce) the attacker picks from previous values. A natural mitigation is to detect that such a value is replayed. This may be done thanks to computationally and memory efficient techniques such as Bloom filters [7, 13].
Another option is to use a counter (instead of pseudo-randomly generating the DevNonce and AppNonce values) which allows to reject already used values in a straightforward manner. This countermeasure has to be implemented both by the device and the NS.

5.1.2 Generate values with no repetition

Another approach to thwart the "replay or decrypt" attack could be to increase the size of DevNonce and AppNonce values so that the birthday bound is unlikely reached, hence preventing the values to repeat. Yet this is crippling since it breaks the compatibility between patched and unmodified equipment. Using a permutation π on $\{0, \ldots, r-1\}$ where r is the number of all possible values ($r = 2^{16}$ for DevNonce and $r = 2^{24}$ for AppNonce) is an alternative way. In the special case where $\pi = \mathsf{id}_r$ then the value is simply a counter.

5.1.3 Verify if a response is bound to a request

The DoS attack against the device relies on the fact that the device blindly accepts any Join Accept message it receives (as long as it is cryptographically valid). Hence the device must be able to discriminate the responses and to verify which one is bound to the request it has sent. Thus a field of the latter (DevNonce, MIC, the whole Join Request message) can be used to compute a field of the former (AppNonce, DevAddr, MIC). Computing the authentication tag on the Join Accept message as well as on the DevNonce value is an option, yet it breaks the compliance between equipment. Instead we propose to compute the DevAddr parameter the following way. Let NwkAddr be the least 25 significant bits. NwkAddr is computed as $\texttt{NwkAddr} = \mathsf{H}(\texttt{DevNonce}, \texttt{AppNonce}, \texttt{DevEUI})$ where H is a collision-resistant function. Using as supplementary input the device's identifier DevEUI allows to discriminate two Join Accept messages in the case the same root key AppKey is shared by several devices. The addition of the parameter AppNonce in the calculation aims to involve the NS. Moreover this formula allows to diversify the DevAddr values among different sessions.

For a given device (*i.e.*, a fixed DevEUI value), a

Table 2: Countermeasures to mitigate the attacks against LoRaWAN (D: device. X^*: X is optional depending on the technical choice for the countermeasure.)

To be implemented by	Attack / Countermeasure	(A1) Replay or decrypt (D, §3.1.1)	(A2) Replay or decrypt (NS, §3.1.2)	(A3) DoS (D, §3.2.1)	(A4) DoS (NS, §3.2.2)
D and NS*	(C1) Detect a replay of `AppNonce`, `DevAddr`	●		●	
NS and D*	(C2) Detect a replay of `DevNonce`		●		●
D	(C3) Generate `DevNonce` values with no repetition	●		●	
NS	(C4) Generate `AppNonce` values with no repetition		●		●
D and NS	(C5) Verify that the received Join Accept message corresponds to the sent Join Request message			●	
NS	(C6 (NS)) Verify that the session keys are shared				●
D	(C6 (D)) Verify that the session keys are shared			●	

`DevAddr` value computed that way depends only on the `DevNonce` and `AppNonce` values. Therefore, if `DevNonce` repeats either "naturally" (birthday paradox) or under coercion (*e.g.*, "replay or decrypt" attack), another Join Accept message than the one actually sent by the NS may be cryptographically valid. Thus this countermeasure (C5 in Table 2) must come with the guarantee that either the `DevNonce` values do not repeat (C3), or the device is able to detect a replay of `AppNonce` values (C1).[18]
This countermeasure based on the `DevAddr` calculation must be implemented by both the device and the NS.

5.1.4 Key confirmation

Each key exchange in LoRaWAN leads to a conundrum with regard to whether the device and the NS actually share the same session keys. In order to eventually solve the riddle both peer may exchange, as soon as the new keys are computed, a protected frame and verify its authentication tag. That frame could be a so called "confirmed" frame requiring an acknowledgment (without any applicative payload though), or a command requiring a response (*e.g.*, *LinkCheckReq* sent by the device, or *DevStatusReq* sent by the NS).

Of course if the same session keys are computed twice (*e.g.*, through a "replay or decrypt" attack), the same frames can be replayed to confirm the key exchange. Therefore this countermeasure (C6 (D), C6 (NS) in Table 2) must come with another one aiming at precluding the reuse of previous security parameters (respectively C1 or C3, and C2 or C4).

[18]Generating `AppNonce` values with no repetition is not enough since an attacker could still replay such a value.

5.2 Recommended countermeasures

The reduced LoRaWAN parameters size limits the efficiency of some countermeasures we propose by paving the way to new attacks. Indeed, generating a parameter (`DevNonce`, `AppNonce`) with no repetition (C3, C4), and detecting replays (C1, C2) are some countermeasures we propose. Yet applying one of these methods *while keeping at the same time* the original parameter size (for compliance reasons) may allow to exhaust all possible `DevNonce` or `AppNonce` values. This can be done if an attacker succeeds in compelling the device or the NS to reach the maximum parameter value (if the parameter is a counter), or to record all values (if replays are detected).[19] Therefore the methods to be implemented in order to thwart the attacks against LoRaWAN must be chosen with caution. We recommend to implement the following.

Apply C4 This countermeasure aims at thwarting attack A2. A counter may be used to produce the `AppNonce` values. The counter must not overlap, and one different counter should be used for each device in order not to artificially lower the number of activations per device.

C4 has to be implemented by the NS.

Apply C1 This countermeasure aims at thwarting attack A1. It may be implemented using computationally and memory efficient techniques such as Bloom filters. However the `AppNonce` parameter being a counter (C4), it is enough for the device to store the last received

[19]See Appendix A for details.

AppNonce value in order to detect a replay.

C1 has to be implemented by the device.

Do not apply C2 The NS must not keep track of all DevNonce values. Hence an attack aiming at exhausting all possible DevNonce values and targeting the NS is avoided (see Section A).

Do not apply C3 The device must not generate DevNonce values in such a way as to guarantee their uniqueness (keep using pseudo-random values). Hence an attack aiming at exhausting all possible DevNonce values and targeting the device is avoided (see Section A).

Apply C5 This countermeasure aims to check that the Join Request and Join Accept messages are bound in order to thwart attack A3. We recommend to compute the DevAddr parameter the following way. Let NwkAddr be the least 25 significant bits. NwkAddr is computed as $\mathtt{NwkAddr} = \mathsf{H}(\mathtt{DevNonce}, \mathtt{AppNonce}, \mathtt{DevEUI})$ where H is a collision-resistant function.

C5 has to be implemented both by the device and the NS.

Apply C6 (NS) This countermeasure aims at thwarting attack A4. We suggest to implement it the following way. Straight after the key exchange is done, the NS must send a *DevStatusReq* command and verify (authentication tag) the *DevStatusAns* response from the device, or verify, if it comes earlier, the first frame sent by the device. The lack of response must be read into this as an issue (device or NS under attack).

In addition the NS must keep all sets of session keys from the last valid one up to the latest computed one. When the NS receives an uplink frame (carrying a *DevStatusAns* response, or another uplink frame), it checks the authentication tag with all keys, starting from the latest. If the keys that match with the authentication tag belong to one of the (currently) unapproved sets, then the NS keeps this set of session keys only an drops all the others. This set becomes then the last valid one.

C6 has to be implemented by the NS.

These countermeasures are recommended if both NS and devices can be modified. If not, we recommend the following. Regarding the NS, apply all recommended countermeasures above.
Regarding the device, apply the following countermeasures instead.

Apply C1 This countermeasure aims at thwarting attack A1. The device must keep track of all the AppNonce values it receives (*e.g.*, using Bloom filters). When the device receives a Join Accept message it must verify the AppNonce parameter and reject the message if the AppNonce is a reused value.

Apply C6 (D) This countermeasure aims at thwarting attack A3. Straight after the key exchange is done (Join procedure), the device must send a *LinkCheckReq* command (with no frame payload) and verify (authentication tag) the *LinkCheckAns* response from the NS, or verify, if it comes earlier, the first frame sent by the NS. If the *LinkCheckAns* response is not valid or if no valid downlink frame is received this must be read into an issue (device under attack).

6 Related work

Few analyses on LoRaWAN have been done and publicly released. Most of the public reviews deal with technical consideration such as the network management (secret keys storage, *etc.*) and generic attacks (*e.g.*, hardware attacks, web attacks) unrelated to the LoRaWAN protocol. Some attacks, which exploit specific features of the protocol, are mentioned but without excess of details.

Regarding the presentation [24], no paper nor slides were made publicly available after the conference (to the best of our knowledge), however we got a summary of the talk. Yet we cannot claim to be aware of all the specifics provided during the talk.

DoS against a device Lifchitz notes that, since the DevNonce parameter is pseudo-random, a reuse is possible due to the birthday paradox [24].

According to L'Héréec and Joulain, a way to perform a DoS attack is to flood the device with repeated Join Accept messages [23].

The previous authors and Miller note that an alternative way to attack the device is to replay to the NS a previous Join Request message (because the server keeps track of a "*certain number of DevNonce values*"), leading to the device "disconnection" from the network [24, 23, 30].

According to Tomasin, Zulian, and Vangelista, a device may be precluded from joining the network after a certain number of sessions, depending on the NS' behaviour [45]. This may happen either "naturally" (if the NS keeps track of all received DevNonce values), or due to an attack (if the NS decides to exclude a device which it repeatedly receives Join Request messages replays from – the replays being sent by an attacker). We note that if the DevNonce value repeats there is a more valuable attack than a DoS (namely the "replay or decrypt" attack described in Section 3.1.1).

Moreover the authors erroneously recommend to use a 2-byte counter for `DevNonce` instead of a pseudo-random value, and to increment the counter only when a (valid) Join Accept message is received. According to the authors the latter ensures that the `DevNonce` counter is shared by the device and the NS. Firstly this is wrong since it is possible to replay any Join Accept message. Secondly this recommendation leads to a simple attack which allows to disconnect the device from the network once and for all. This attack is of the same kind than the one described in Section 3.2.2. The attacker does the following: when the device sends a Join Request message, the attacker forbids (once only) the device from receiving the Join Accept message sent by the NS. Hence the device reuses the same `DevNonce` value in subsequent Join Request messages. And these messages are then discarded by the NS since they carry an invalid (*i.e.*, already received) `DevNonce` value. Therefore the device is stuck since it keeps using the same `DevNonce` value (which is continuously discarder by the NS).

DoS against the NS Miller proposes a denial of service attack against the NS by flooding the server [30, 29].

Frame replay and frame decryption Lifchitz notes that the pseudo-random `AppNonce` parameter may repeat due to the birthday paradox [24]. Hence, under the strong assumption that the `DevNonce` value is "*forced*" (device controlled by an attacker), a keystream reuse is possible with high probability after $\sqrt{2\ln(2)\times 11\times 2^{24}} \simeq 16{,}000$ activations, or 22 hours if a key exchange is done in 5 seconds. In fact such a statement is wrong or, at least, hazy: if both `DevNonce` and `AppNonce` values repeat, this leads to a *session keys* reuse. In order to get a *keystream* reuse, it is necessary for the `DevAddr` parameter to repeat as well. Moreover this means a continuous series of key exchanges without any intermediary application frame. Hence the sake of such an attack may be questioned.

Furthermore it is unclear where the numbers come from. We may assume that the NS keeps track of 10 `DevNonce` values, and the attacker uses 11 different Join Request messages, randomly choosing one at each try. However the NS unlikely accepts every such message since the same message (hence the same `DevNonce` value) is picked after roughly 4 tries. Hence the number of activations needed to get a reuse of both `DevNonce` and `AppNonce` values is higher than the provided number 16,000, as well as the duration of the attack. And the number of 10 stored values seems to be implementation specific. Yet we cannot claim this is the genuine purpose of [24].

Finally this attack is unlikely successful against a NS implementing version 1.0.2 (the current 1.0 version). Indeed, according to the specification, the NS must receive a valid uplink frame protected by the new security parameters before dropping the current ones and using the new ones. The attack leads to the computation of the same session keys two different times. Yet, with high probability, these keys are fresh (*i.e.*, never used previously by the NS with a legitimate device) because the attacker has no control on the `AppNonce` parameter. This means that the attacker has to forge a valid uplink frame if she wants to compel the NS to use these keys. That is the attacker must forge a valid 32-bit authentication tag (without the corresponding key). That being said, we are not aware of the LoRaWAN version analysed in that talk (1.0.1 or 1.0.2).

Authentication tag forgery The application frames are protected with a 4-byte authentication tag, hence, according to Miller, forgery attempts may be tried mainly against the NS [30, 29].

Random bit generation Tomasin *et al.* show also that it is possible to make the distribution of the random bit generator output produced by a device (hence the distribution of the `DevNonce` values) deviate by influencing on the signal strength [45].

7 Conclusion

The extensive analysis we perform of the security protocol LoRaWAN 1.0 shows that it suffers from several weaknesses. We describe precisely how these flaws can be exploited to carry out attacks, including practical ones. These attacks lead to a breach in the network availability, data integrity, and data confidentiality.

The first type of attacks ends up with the device disconnection from the network. The second kind allows an attacker to replay and to decrypt frames, hence to deceive the NS (or the AS) or the device (which may be an actuator). We emphasise that the aforementioned attacks, due to the protocol flaws, do not lean on potential implementation or hardware bugs, and are likely to be successful against any equipment implementing LoRaWAN 1.0. Likewise the attacks do not entail a physical access to the targeted equipment and are independent from the means used to protect secret values (*e.g.*, using a tamper resistant module such as a Secure Element).

We present new attacks and, contrary to previous works (to the best of our knowledge), the attacks we describe target both types of equipment (device or NS). Moreover our attacker needs only to act on the air interface (to eavesdrop and send data), but she does not need to get a physical access to any equipment (in particular the device).

In addition we provide practical recommendations allowing to thwart the attacks we have found, while at the same time being compliant with the specification, and keeping the interoperability between patched and unmodified equipment. According to us, the recommended countermeasures can be implemented in a straightforward manner.

Furthermore the attacks we describe may allow to appreciate the security properties provided by the upcoming version 1.1 of LoRaWAN, and to bridge the gap if these properties do not fulfill their intended purpose.

A Exhaustion attack

Generating a parameter (`DevNonce`, `AppNonce`) with no repetition (C3, C4), and detecting replays (C1, C2) are some countermeasures we propose. Yet, in LoRaWAN, size does matter. Applying one of these methods *while keeping at the same time* the original parameter size (for compliance reasons) may lead to an attack aiming at exhausting all possible `DevNonce` or `AppNonce` values, hence forbidding the NS or the device to start a new activation. Therefore this exhaustion attack, targeting the device or the NS it connects to, may lead to an irrevocable disconnection of the device.

A.1 Against the `DevNonce` parameter

Core If the device generates `DevNonce` values with no repetition (C3) or if the NS keeps track of all `DevNonce` values it receives (C2), it is possible to disconnect the device once and for all.

Attack Every time the device sends a Join Request message, the attacker replies with a "false" Join Accept message. Hence the device generates a new message once again. If countermeasure C3 is applied, all `DevNonce` values will be eventually used. If countermeasure C2 is applied, the NS will refuse further Join Request messages once all possible `DevNonce` values have been received, be these values pseudo-random or not.

Numerical sample Let us assume that a key exchange is done in 5 seconds. If the `DevNonce` values never repeat, the attack targeting the device is achieved in 91 hours.

Let us consider the case when the NS keeps track of all `DevNonce` values. If the values are pseudo-random, the proportion effectively generated by the device, hence received by the NS after ℓ key exchanges, is $p = 1 - \exp(-\frac{\ell}{2^{16}})$. For this proportion to be $p = 99\%$, the number of key exchanges must be at least $\ell = -2^{16} \times \ln(1 - p)$. This corresponds to $\ell \simeq 301{,}804$ activations and more than 17 days to exhaust almost all `DevNonce` values.

A.2 Against the `AppNonce` parameter

Core If the NS generates the `AppNonce` parameter so that it never repeats (C4), or if the device keeps track of all `AppNonce` values it receives (C1), then it is possible to disconnect the device once and for all.

Attack Let us consider the first case (C4). The purpose is to compel the NS to use all possible `AppNonce` values. The NS generates a Join Accept message (hence a new `AppNonce` value) only if it receives a valid Join Request message. Therefore the NS must accept as many Join Request messages as possible `AppNonce` values. Since $|\texttt{DevNonce}| < |\texttt{AppNonce}|$, this is possible only if the NS does not keep track of all `DevNonce` values it receives (namely if the NS does not apply C2, which is likely its behaviour). Then the attacker can use a circular list of Join Request messages. Such messages can be collected using the technique described in Section 3.1.1, and then used in a similar way than the one described in Section 3.1.2.
Note that if the NS uses the same pool of `AppNonce` values for all the devices, this leads to the definitive disconnection of all these devices. In such a case the attack may be distributed among several "false" devices (controlled by the attacker; no duty cycle enforced).

Let us consider the second case (C1). The purpose of this attack is to make the device keep track, hence receive, all possible `AppNonce` values. This means that the NS has to accept as many Join Request messages as possible `AppNonce` values. Therefore the NS must not keep track of all the `DevNonce` values it receives (since $|\texttt{DevNonce}| < |\texttt{AppNonce}|$). Namely the NS must not apply C2. Yet this is not sufficient. Indeed the device accepts as many Join Accept messages (hence `AppNonce` values) as Join Request messages it sends. Therefore if the device generates `DevNonce` values with no repetition, it limits the number of received `AppNonce` values. Hence the device must not apply C3. Therefore this attack is possible if the NS does not apply C2 and the device does not apply C3 (which is likely their basic behaviour).

Moreover the implementation of this attack implies to be able to compel the device to send multiples Join Request messages *while receiving* the corresponding Join Accept responses. We have not identified such means but to be able to influence on the device power supply. Yet, if the device is switched off, it may lose memory of the stored `AppNonce` values, which is orthogonal to the goal of this attack.

Numerical sample If the AppNonce values do not repeat (C4), they are all produced after $2^{24} \times 5$ seconds $= 2.66$ years (using one device).
If the same pool of AppNonce values is used by the NS for all devices, the attack may be distributed among several devices controlled by the attacker. If 300 such devices are used in parallel, the attack is achieved in 3 days approximately.

If the device keeps track of all AppNonce values (C1), and if the values are pseudo-random, $p = 99\%$ values are received by the device after $\ell = -2^{24} \times \ln(1-p) \simeq 77.26 \times 10^6$ activations. This means more than 12 years to exhaust almost all AppNonce values.

References

[1] Al Fardan, N. J., and Paterson, K. G. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *2013 IEEE Symposium on Security and Privacy* (May 2013), SP '13, pp. 526–540.

[2] Bellare, M. Practice-Oriented Provable-Security. In *Proceedings of the First International Workshop on Information Security* (1997), E. Okamoto, G. Davida, and M. Mambo, Eds., vol. 1396 of *LNCS*, Springer-Verlag, pp. 221–231.

[3] Bellare, M., Desai, A., Jokipii, E., and Rogaway, P. A Concrete Security Treatment of Symmetric Encryption. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science* (1997), FOCS '97, IEEE, pp. 394–403.

[4] Bellare, M., Kilian, J., and Rogaway, P. The Security of the Cipher Block Chaining Message Authentication Code. *Journal of Computer and System Sciences 61*, 3 (Dec. 2000), 362–399.

[5] Bellare, M., and Namprempre, C. Authenticated Encryption: Relations Among Notions and Analysis of the Generic Composition Paradigm. *Journal of Cryptolology 21*, 4 (Sept. 2008), 469–491.

[6] BiztechAfrica. FastNet announces Africa's first dedicated M2M network and IoT developer academy, October 2015. `http://www.biztechafrica.com/article/fastnet-announces-africas-first-dedicated-m2m-netw/10718/`.

[7] Bloom, B. H. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM 13*, 7 (July 1970), 422–426.

[8] Blutetooth SIG. Bluetooth specification. Available via `https://www.bluetooth.com/specifications/adopted-specifications`.

[9] Briodagh, K. Japan Opens New LoRaWAN Network for IoT Testing, July 2016. `http://www.iotevolutionworld.com/iot/articles/423324-japan-opens-new-lorawan-network-iot-testing.htm`.

[10] Canvel, B., Hiltgen, A., Vaudenay, S., and Vuagnoux, M. Password interception in a SSL/TLS channel. In *Advances in Cryptology* (2003), D. Boneh, Ed., vol. 2729 of *CRYPTO 2003*, Springer-Verlag, pp. 583–599.

[11] Degabriele, J. P., Paterson, K., and Watson, G. Provable Security in the Real World. *IEEE Security and Privacy 9*, 3 (May 2011), 33–41.

[12] Diffie, W., and Hellman, M. E. Privacy and Authentication: An Introduction to Cryptography. *Proceedings of the IEEE 67*, 3 (Mar. 1979), 397–427.

[13] Dillinger, P., and Manolios, P. Bloom Filters in Probabilistic Verification. In *Formal Methods in Computer-Aided Design* (2004), vol. 3312 of *LNCS*, Springer, pp. 367–381.

[14] Fearn, N. Orange deploys LoRa network in thousands of French towns, September 2016. `https://internetofbusiness.com/orange-lora-network-france/`.

[15] Fischlin, M., Günther, F., Schmidt, B., and Warinschi, B. Key Confirmation in Key Exchange: A Formal Treatment and Implications for TLS 1.3. In *2016 IEEE Symposium on Security and Privacy (SP)* (May 2016), IEEE, pp. 452–469.

[16] Institute of Electrical and Electronics Engineers. IEEE Standard for Local and metropolitan area networks – Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs), Nov. 2011. IEEE Std 802.15.4-2011, revision of IEEE Std 802.15.4-2006.

[17] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Lucky 13 Strikes Back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2015), ASIA CCS '15, ACM, pp. 85–96.

[18] Iwata, T., and Kurosawa, K. OMAC: One-Key CBC MAC. In *Fast Software Encryption* (2003), T. Johansson, Ed., vol. 2887 of *LNCS*, Springer-Verlag, pp. 129–153.

[19] Iwata, T., and Kurosawa, K. OMAC: One-Key CBC MAC – Addendum, Mar. 2003. Available via `http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/omac/omac-ad.pdf`.

[20] Iwata, T., and Kurosawa, K. Stronger Security Bounds for OMAC, TMAC, and XCBC. In *Progress in Cryptology – INDOCRYPT 2003* (2003), T. Johansson and S. Maitra, Eds., vol. 2904 of *LNCS*, Springer-Verlag, pp. 402–415.

[21] Kim, G. Tata to deploy LoRa network for IoT, June 2016. `http://spectrumfutures.org/tata-to-deploy-lora-network-for-iot/`.

[22] Kinney, S. 100 US cities covered by Senet LoRa network for IoT, June 2016. `http://www.rcrwireless.com/20160615/internet-of-things/100-u-s-cities-covered-senet-lora-network-iot-tag17`.

[23] L'Héréec, F., and Joulain, N. Sécurité LoRaWAN. In *Computer & Electronics Security Applications Rendez-vous – C&ESAR* (2016).

[24] Lifchitz, R. Security review of LoRaWAN networks. In *Hardwear.io* (2016).

[25] LoRa Alliance Technical committee. LoRaWAN Regional Parameters, July 2016. LoRa Alliance, version 1.0.

[26] LORIOT.io. `https://www.loriot.io`.

[27] Marek, S. SK Telecom & KPN Deploy Nationwide LoRa IoT Networks, July 2016. `https://www.sdxcentral.com/articles/news/sk-telecom-kpn-deploy-nationwide-lorawan-iot-networks/2016/07/`.

[28] Mason, J., Watkins, K., Eisner, J., and Stubblefield, A. A Natural Language Approach to Automated Cryptanalysis of Two-time Pads. In *Proceedings of the 13th ACM Conference on Computer and Communications Security* (2006), CCS '06, ACM, pp. 235–244.

[29] Miller, R. LoRa Security – Building a Secure LoRa Solution, Mar. 2016. Whitepaper, MWR Labs. Available via `https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-LoRa-security-guide-1.2-2016-03-22.pdf`.

[30] Miller, R. LoRa the Explorer – Attacking and Defending LoRa Systems. In *Information Security Conference – SyScan360* (2016). Available via `https://www.syscan360.org/slides/2016_SG_Robert_Miller_LoRa_the_Explorer-Attacking_and_Defending_LoRa_systems.pdf`.

[31] NANDI, M. Improved Security Analysis for OMAC as a Pseudo Random Function. *Journal of Mathematical Cryptology 3*, 2 (Aug. 2009), 133–148.

[32] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. NIST FIPS 197 Specification for the Advanced Encryption Standard (AES), Nov. 2001. Available via `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[33] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. NIST Special Publication 800-38A Recommendation for Block Cipher Modes of Operation – Methods and Techniques, Dec. 2001. Available via `http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf`.

[34] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. NIST Special Publication 800-38C Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality, May 2004. Available via `http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C_updated-July20_2007.pdf`.

[35] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. NIST Special Publication 800-38B Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication, May 2005. Available via `http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf`.

[36] NEISSE, R., STERI, G., AND BALDINI, G. Enforcement of Security Policy Rules for the Internet of Things. In *2014 IEEE 10th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)* (Oct 2014), pp. 165–172.

[37] SEMTECH. `http://iot.semtech.com`. Last consulted May 15, 2017.

[38] SEYS, S., AND PRENEEL, B. Power Consumption Evaluation of Efficient Digital Signature Schemes for Low Power Devices. In *IEEE International Conference on Wireless And Mobile Computing, Networking And Communications* (August 2005), vol. 1 of *WiMob 2005*, IEEE, pp. 79–86.

[39] SMARTCITIESWORLD. IoT connectivity for 100 million homes in China, November 2016. `https://smartcitiesworld.net/news/news/iot-connectivity-for-100-million-homes-in-china-1139`.

[40] SMARTCITIESWORLD. LoRaWAN IoT network deployed in Japan, September 2016. `https://smartcitiesworld.net/connectivity/connectivity/lorawan-iot-network-deployed-in-japan`.

[41] SMARTCITIESWORLD. Semtech LoRa chosen for new IoT network in New Zealand, September 2016. `https://smartcitiesworld.net/news/news/semtech-lora-chosen-for-new-iot-network-in-new-zealand-949`.

[42] SONG, J. H. AND POOVENDRAN, R. AND LEE, J. AND IWATA, T. The AES-CMAC Algorithm. RFC 4493, June 2006. Available via `https://tools.ietf.org/html/rfc4493`.

[43] SORNIN, N. AND LUIS, M. AND EIRICH, T. AND KRAMP, T. AND HERSENT, O. LoRaWAN Specification, July 2016. LoRa Alliance, version 1.0.2.

[44] THE THINGS NETWORK. `https://www.thethingsnetwork.org`.

[45] TOMASIN, S., ZULIAN, S., AND VANGELISTA, L. Security Analysis of LoRaWAN Join Procedure for Internet of Things Networks. In *IEEE Wireless Communications and Networking Conference Workshops* (2017), WCNCW 2017, IEEE, pp. 1–6.

[46] Z-WAVE ALLIANCE. Z-Wave specification. Available via `http://z-wave.sigmadesigns.com/design-z-wave/z-wave-public-specification/`.

[47] ZIGBEE ALLIANCE. ZigBee specification. Available via `http://www.zigbee.org/download/standards-zigbee-specification/`.