Formal Proof of a Vulnerability in Z-Wave IoT Protocol*

Mario Lilli[a], Chiara Braghin[b] and Elvinia Riccobene[c]
Computer Science Department, Università degli Studi di Milano, Italy

Keywords: Z-Wave Protocol, IoT Security, MITM, Formal Verification, Abstract State Machine, ASMETA.

Abstract: Nowadays, IoT (Internet of Things) devices are becoming part of our daily life. Unfortunately, many of them do not use standardized communication protocols with a provable security guarantee. The use of formal methods is, therefore, highly demanded in order to perform property verification and to prevent possible threats and accidents to users. In this paper, we propose a formal verification of the Z-Wave protocol, claimed to be one of the most secure IoT communication protocols thanks to the new S2 Security class, recently added. Specifically, our analysis targets the joining procedure of a device to the Z-Wave net. We exploit the ASMETA formal framework to model the protocol and to perform formal analysis in terms of model validation against informal documented requirements and verification of the protocol correct behaviour with respect to its security goals. The verification process revealed a vulnerability that could be used to perform a successful Man-In-The-Middle (MITM) attack compromising the secrecy of the exchanged symmetric keys.

1 INTRODUCTION

Nowadays, millions of IoT (Internet of Things) devices (e.g., alarms, door locks, lights, sensors, etc.) are becoming part of our daily life. They are used in very different applications that in many cases are security and safety *critical* services, as for example gas monitoring, home door opening, etc. Installing vulnerable or unsafe devices might have serious consequences in terms of user's privacy and safety, and many alliance of corporations are adopting security procedures to protect the traffic generated by their protocols. Nevertheless, the approaches adopted to meet security requirements have often proven to be insufficient, and many of them do not use standardized communication protocols with provable guarantee of security and safety, which can be achieved only by means of formal verification (Hofer-Schmitz and Stojanović, 2020).

Among the plethora of IoT security protocols, the Z-Wave protocol (Z-wave, 2020) is considered the most promising in terms of security features offered. It is a communication protocol designed primarily for home automation, and security is provided by message encryption. It was proprietary for long time, and this is the reason why the protocol has not been largely investigated in literature. It became publicly available only in 2017, then it has been updated with a new security layer (S2 Security), overcoming the shortcomings of the previous S0 security layer which was found vulnerable (Fouladi and Ghanoun, 2013).

In this paper, we propose a formal verification of the Z-Wave protocol with S2 security layer in terms of the Abstract State Machine formal method (Börger and Raschke, 2018; Börger and Stärk, 2003) and we exploit the validation and verification techniques offered by the supporting framework ASMETA (Arcaini et al., 2011). More precisely, our protocol analysis targets the *joining procedure* of a device to the Z-Wave network, which involves the use of S2 security class. It defines the key establishment phase, where a new joining device negotiates with the controller the symmetric keys that it will use to secure all communications with the other nodes of the network.

In order to verify if the protocol security goals are satisfied, we also formalised a malicious adversary, able to intercept and craft messages. Then, the formal verification of the joining procedure revealed a feasible Man-In-The-Middle (MITM) attack. Indeed, the model checker returned a path showing how an attacker obtains all the encryption keys exchanged during the joining procedure. We reported the vulner-

[a] https://orcid.org/0000-0001-7236-9171
[b] https://orcid.org/0000-0002-9756-4675
[c] https://orcid.org/0000-0002-1400-1026
*The work was partially supported by the SEED Project "Situational awareness in critical Infrastructure Environment (SENTINEL)", Università degli Studi di Milano.

Lilli, M., Braghin, C. and Riccobene, E.
Formal Proof of a Vulnerability in Z-Wave IoT Protocol.
DOI: 10.5220/0010553301980209
In *Proceedings of the 18th International Conference on Security and Cryptography (SECRYPT 2021)*, pages 198-209
ISBN: 978-989-758-524-1

ability to Silicon labs and Z-Wave Alliance. They confirmed that the attack works for one authentication method, is previously unknown, and is feasible in practice. To avoid the exploitation of the flaw, an improved version of the protocol, called *SmartStart*, employs a QR code scan of the DSK (Device Specific Key) that renders the attack infeasible by removing human operator error from the attack scenario.

The contribution of the paper is threefold:

- A rigorous and precise description of the Z-Wave protocol with S2 security layer, overcoming the limitation of a fragmented and unclear documentation of the protocol.
- A scenario-based validation to check our model against the protocol requirements.
- Verification of the protocol security goals with the detection of a MITM attack (confirmed by protocol designers).

The rest of the paper is organised as follows. In Sect. 2 we briefly review existing results on formal methods in the context of IoT security protocols. Sect. 3 presents in a concise way the theory behind the ASM formal method and the ASMETA framework with its analysis approaches; it also introduces the operation of Z-Wave. Sect. 4 presents the ASM formal modeling, validation and verification of the protocol, together with the attacker model and the description of a feasible MITM attack. Sect. 5 concludes the paper and outlines future research work.

2 RELATED WORK

Specification and verification of IoT protocols is a recent topic in the context of formal methods, and small literature exists in this area. A comprehensive review on formal methods for IoT protocols is given in (Hofer-Schmitz and Stojanović, 2020). The paper gives details both on the properties taken in consideration, and on the methods applied. Moreover, they identify four areas for the application of formal methods: *i*) a functional area, concerning the verification of protocol stack and key management; *ii*) a second one, regarding the extension of the protocol with provable enhanced security features; *iii*) a third area on the evaluation of security properties; and *iv*) the last one on the implementation verification. Among IoT protocols, the mostly analised are Zigbee, LoRaWAN, Bluetooth, Narrowband IoT, 6LoWPAN, 5G. Z-Wave has small literature since it has been proprietary for a long time. To the best of our knowledge, the only contribution on formal methods applied to Z-Wave protocol is that in (Mohsin et al., 2017). It investigates the possibility of flaws generated by wrong configurations in Z-Wave scenes.

The Zigbee protocol is analyzed in (Melaragno et al., 2012) by using AVISPA and Casper, where the existence of a known security flaw is discovered. AVISPA, in combination with the graphical tool SPAN, allows a sort of protocol runtime verification where Message Sequence Charts are built during the protocol execution and are used to check if the execution performs consistently with the expected message exchange (Armando et al., 2005; Viganò, 2006).

Two versions of LoRaWAN are compared in (Eldefrawy et al., 2019) using models specified with Scyther: for LoRaWAN v1.0, they find some flaws already known in the literature, while they do not find any evident vulnerability in v1.1. However, they claim that this is possibly due to the limitation of Scyther in terms of verification capabilities and the Dolev-Yao assumption of perfect cryptography (Dolev and Yao, 1983), under which the protocol analysis is conducted.

Some of the Bluetooth protocol versions have been verified with formal methods. Version v5.0-Part I has been checked by Sun and Sun (Sun and Sun, 2019); they perform a formal analysis of the secure simple pairing (SSP) that constitutes a considerable part of the security in home automation scenarios. Four different models are used depending on the capability of the device, but they all resist passive eavesdropping and MITM attacks.

Two schemas for the 6LoWPAN protocol have been verified. (Qiu and Ma, 2016) proposes a secure Proxy Mobile IPv6 (MPIPv6) that prevents some attacks, including replay attacks, man-in-the-middle attacks, privileged insider attacks and Sybil attacks. The attacks and the schema are executed by using AVISPA and a Java simulation. The work in (Qiu and Ma, 2018) presents an extension schema of 6LoWPAN, which grants the CIA (Confidentiality, Integrity and Authenticity) for a group of resource-constrained 6LoWPAN devices. A method based on the BAN logic and the tool Scyther proves the resistance of the schema against replay, MITM, impersonation, privileged insider, and Sybil attacks.

(Basin et al., 2018) provides a formal analysis of the 5G authenticated key exchange (AKA) protocol. They formalise the 5G AKA with Tamarin, and their verification shows possible privacy flaws. They propose a provable fix for the vulnerability found.

Although the results on formal analysis of the Z-Wave protocol are very limited, the security of the Z-Wave protocol has been instead investigated and tested in practice, by means of test beds in real environments. In (Unwala et al., 2018), Z-Wave is analysed with respect to common attacks to IoT systems. In (Badenhop et al., 2017), the frame forwarding and topology management aspects of a previous version of the Z-Wave routing protocol are reverse engineered. Then, a security analysis is also performed on the network and the possibility of modifying the topology and routes by an outsider is found. In (Kim et al., 2020), the authors propose three different attack vectors that, if combined, can cause critical damage (e.g., DoS).

3 BACKGROUND

3.1 Z-Wave Protocol

Z-Wave is a wireless radio frequency based communications protocol designed primarily for home automation, using frequencies from 865 to 926 MHz. Nodes can be residential appliances or other devices, such as lighting control, thermostats, security systems, windows, locks and garage door openers.

A Z-Wave network is a mesh network using low-energy radio waves to communicate: nodes within range communicate directly with one another, remote nodes rely on intermediate nodes (see Fig. 1). Nodes can be either *controllers* or *slaves*: controllers are responsible for including and excluding nodes, and for maintaining the routing table of the whole network, whereas slaves are the real smart things of the network composed of lights, switches, dimmers, sensors, motors and other. In order to limit battery consumption, nodes are alternatively active or in sleep mode. Thus, only some nodes can relay a message.

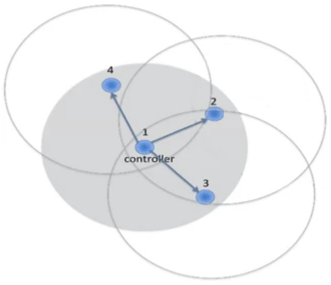

Figure 1: Z-wave network topology.

Z-Wave has been designed with security in mind: security is achieved by encrypting the messages exchanged between devices. The first version of the security layer (S0 Security) has been found vulnerable due to a weak temporary key (Fouladi and Ghanoun, 2013) during the joining phase. The new Z-Wave security layer (S2 Security) evolves from S0 and provides messages integrity, confidentiality, authentication and data freshness, by using a combination of symmetric encryption and message authentication code (MAC) (as shown in Tab. 1).

Table 1: Security features of Z-Wave protocol.

Security Feature	Technique
Confidentiality Authentication	AES CCM mode 128 bit key
Integrity	AES CMAC mode 128 bit key
Freshness	Pre-Agreed Nonces (PAN) Multicast Pre-Agreed Nonces (MPAN)

Each device has a role, and each role is linked to one or more security classes by design. There are four possible security classes: S2-AccessControl, S2-Authenticated, S2-Unauthenticated and S0 for retro compatibility. In general, door locks, garage door openers and controllers fall at least into S2-AccessControl class, whereas all types of secure end-devices such as window blind motors, switchers, other sensors and secondary controllers belong to S2-Authenticated class. During the *joining procedure*, a new device joining the network first negotiates with the controller the subset of its security classes that the controller can handle. Then, the controller sends one key for each security class it has been granted. In this way, a Z-Wave network is segmented by grouping nodes into security classes that communicate securely using the same AES 128 bits key. The usage of different keys for each class protects the network when one of the granted keys is compromised, by reducing the number of devices using the compromised key.

Since AES-128 symmetric keys are used to secure all communication between nodes, the key establishment phase, i.e., the *joining procedure* occurring when a new device joins the network, is the core of S2 Security layer. The procedure works in two steps (see Fig. 2): during the first phase, the two participants agree on a temporary AES key to be used in the second phase of the protocol to encrypt the symmetric keys granted to the new device. As a consequence, key establishment relies on the secrecy of the temporary key K_t: if the key is compromised, all the symmetric keys exchanged in the second part of the protocol are compromised. Therefore, in the following, we discuss and analyse only the messages exchanged during the first phase of the protocol, and that can be divided into three subsections:

1. A first phase starting with the device announcing its device type with a *Node Information Frame* (NIF). Subsequently, controller *A* and slave *B* agree on which security classes to share, by ex-

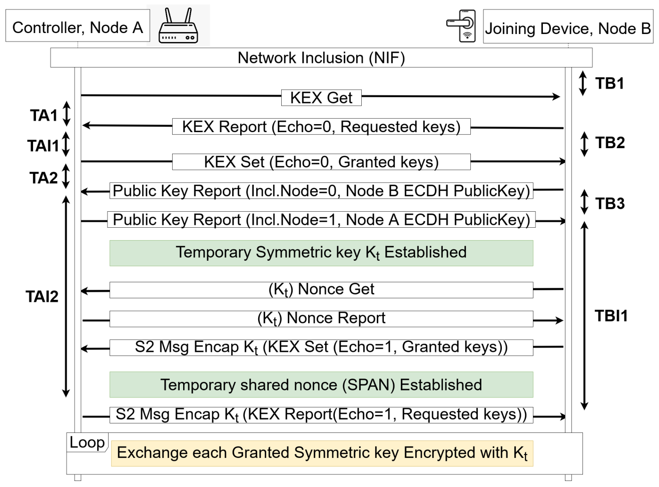
Controller, Node A
Joining Device, Node B
Network Inclusion (NIF)
TB1
KEX Get
TA1
KEX Report (Echo=0, Requested keys)
TAI1
TB2
KEX Set (Echo=0, Granted keys)
TA2
Public Key Report (Incl.Node=0, Node B ECDH PublicKey)
TB3
Public Key Report (Incl.Node=1, Node A ECDH PublicKey)
Temporary Symmetric key K_t Established
TAI2
(K_t) Nonce Get
(K_t) Nonce Report
TBI1
S2 Msg Encap K_t (KEX Set (Echo=1, Granted keys))
Temporary shared nonce (SPAN) Established
S2 Msg Encap K_t (KEX Report(Echo=1, Requested keys))
Loop
Exchange each Granted Symmetric key Encrypted with K_t

Figure 2: Z-Wave *joining procedure*.

changing KEX Get, KEX Report and KEX Set messages.

2. A second phase, where the two participants exchange their public keys, used to generate the temporary encryption key by means of Elliptic Curve Diffie-Hellman (ECDH) on Curve25519 with a public key length of 256 bits. ECDH is a key agreement protocol that allows two parties A and B, both owning an elliptic-curve public-private key pair, to establish a shared secret over an insecure channel. Actor A computes the shared secret with her/his own private key and B's public key, while B computes the same shared secret with B's private key and A's public key. No one but A or B can compute the shared secret, unless (s)he can solve the elliptic curve discrete logarithm problem.

   Since ECDH is vulnerable to a man-in-the-middle attack, Z-Wave Alliance introduced (Z-Wave, 2020) three Out-Of-Band (OOB) authentication to prevent eavesdropping and MITM attack vectors. In order to continue the *joining procedure*, the user can use one of the following methods: (1) enter as a PIN the first five digits of the DSK (printed on the device), obfuscated during the RF transmission; (2) verify the full DSK by visual inspection; (3) scan a QR code to verify the full DSK (p. 93, requirement CC:009..01.00.11.05F [1] in (Z-Wave, 2020)).

   The attack presented here applies only to the first method: the PIN code is inserted on the controller side if the slave device is S2 security native (i.e., released with S2 security), or on the slave side if the slave has been updated to S2 security.

3. The third section consists of a challenge-response protocol based on a nonce allowing the controller and slave to check if they both know K_t. They also resend KEX Report and KEX Set messages that are used to assure no message tampering.

The protocol then continues with a loop assigning an AES key for each granted class. Please notice that many sub-phases of the protocol are temporised, thus, if a message does not arrive in time, the slave or controller will abort the *joining procedure* (e.g., timer $TB1$ ends the procedure if message KEX Get takes more than 30 seconds to arrive).

As an example, consider the case of a smart light (node B) with native S2 security, asking to join the network. In a normal execution of the protocol, the light asks two security classes, S2 Unauthenticated and S0, and receives the corresponding symmetric keys. In case the network consists only of native S2 security devices, in order to strengthen the security of its network, the user may disable class S0 security in the controller (node A). In this specific scenario, let us suppose that the KEX Report message sent from B to A has been corrupted. As a consequence, the controller receives the request only of S0 security class. This situation triggers the error message KEX FAIL KEX KEY since the intersection between the requested security classes and the granted security classes is empty. This behaviour follows a security requirement expressed in the protocol specification.

3.2 Abstract State Machines

Abstract State Machines (ASMs) (Börger and Raschke, 2018; Börger and Stärk, 2003) are an extension of Finite State Machines (FSMs).

[1] “The user MUST be prompted a dialog to visually validate the bytes 3..16 of Node B’s DSK”.

ASM *states* replace unstructured FSM control states by algebraic structures, i.e., domains of objects with functions and predicates defined on them. An ASM *location*, defined as the pair (*function-name*, *list-of-parameter-values*), represents the ASM concept of basic object container. The couple (*location*, *value*) represents a memory unit. Therefore, ASM states can be viewed as a set of abstract memories.

State transitions are performed by firing *transition rules*, which express the modification of functions interpretation from one state to the next one and, therefore, they change location values. Location *updates* are given as assignments of the form $loc := v$, where loc is a location and v its new value. They are the basic units of rules construction.

By a limited but powerful set of *rule constructors*, location updates can be combined to express other forms of machine actions as: guarded actions (`if-then`, `switch-case`), simultaneous parallel actions (`par` and `forall`), sequential actions (`seq`), non-deterministic actions (`choose`).

Functions which are not updated by rule transitions are *static*. Those updated are *dynamic*, and distinguished in *monitored* (read by the machine and modified by the environment), *controlled* (read and written by the machine), *shared* (read and written by the machine and its environment).

An ASM model has a predefined structure consisting of: a *signature*, which contains declarations of domains and functions; a block of *definitions* of static domains and functions, transition rules, state invariants and properties to verify; a *main rule*, which is the starting point of a machine computation; a set of *initial states*, one of which is elected as *default* and defines an initial value for the machine locations.

An ASM *computation* (or *run*) is defined as a finite or infinite sequence $S_0, S_1, \ldots, S_n, \ldots$ of states of the machine, where S_0 is an initial state and each S_{n+1} is obtained from S_n by firing the unique *main rule* which in turn could fire other transitions rules.

ASMs allow modeling different computational paradigms, from a *single* agent executing an ASM, to distributed *multiple* agents, which is the computational paradigm we used in our Z-Wave model. A *multi-agent ASM* is a family of pairs $(a, ASM(a))$, where each a of a predefined set *Agent* executes its own machine $ASM(a)$ (specifying the agent's behavior), and contributes to determine the next state by interacting synchronously or asynchronously with the other agents. A predefined function *program* on *Agent* is used to associate the ASM with an agent. Since agents of a same kind (i.e., agents representing protocol slaves) have the same behaviour, within transition rules, each agent can identify itself by means of a special 0-ary function *self* : *Agent* which is interpreted by each agent a as itself.

```
 1  asm zwaveProtocol_join
 2
 3  signature:
 4    domain Slave subsetof Agent
 5    domain Controller subsetof Agent
 6    .....
 7    static nodeA: Controller
 8    static nodeB: Slave
 9    ....
10  definitions:
11    main rule r_Main =
12      par
13        program(nodeA)
14        program(nodeB)
15      endpar
16  default init s0:
17    function slaveState (a in Agent) = if (a = nodeB) then INIT_SLV endif
18    function controllerState (c in Agent) = if (c = nodeA)
19                                                    then INIT_CTRL endif
20      ...
21    agent Controller:
22      r_controllerRule[]
23    agent Slave:
24      r_slaveRule[]
```

Code 1: Excerpt of the ASM model for Z-Wave protocol.

Code 1 shows an excerpt of the *multi-agent ASM* model for the Z-Wave joining procedure between two nodes (A and B), working as controller and as slave, respectively. `r_controllerRule[]` on line 21 is the ASM program associated to an agent of type *Controller*, while `r_slaveRule[]` on line 23 is that of an agent of type *Slave*; *nodeA* and *nodeB* are initiated as the corresponding type of agents.

Tools & Validation and Verification Techniques. The ASM formal method is supported by the toolset ASMETA (ASM mETAmodeling) (Arcaini et al., 2011) for model editing, validation and verification. Model construction, especially when taking the system requirements from natural language, can often be error prone. For this reason, it is essential to be able to *validate a model* against its functional and non functional requirements. ASM models can be validated in terms of model simulation (by using `AsmetaS`), animation (by `AsmetaA`), and scenarios execution (by `AsmetaV`). In the latter case, each scenario contains the expected system behavior, and the tool checks whether the machine runs correctly. It is also possible to *verify properties*, expressed in temporal logic, by means of model checking (with `AsmetaSMV`): the tool will check if the property holds during all possible model executions.

Remark. ASMs offer several advantages as formal method: (1) due to their *pseudo-code format*, they can be easily understood by practitioners and can be used for *high-level programming*; (2) they allow for system specification at any desired *level of abstrac-*

tion; (3) they are *executable models*, so they are suitable also for lighter forms of model analysis such as simple simulation to check model consistency w.r.t. system requirements; (4) they support techniques for mapping models to code (e.g., to C++ (Bonfanti et al., 2020) or Java (Arcaini et al., 2017)); (5) they can be used for *modeling distributed systems*, as IoT networks; (6) the ASMETA framework allow for an integrated use of tools for different forms of model analysis, it is maintained and under continuous features improvement.

4 Z-Wave FORMAL VERIFICATION

In this section, we present the ASM specification of Z-Wave protocol. We started by defining a model of the *joining procedure*, which has been validated by checking that, during the execution, it behaves as required in the protocol informal description. Then, we explicitly modelled an attacker controlling the network, able to delete, inject, modify and intercept any message. The verification phase highlighted a feasible MITM attack, where the attacker is able to obtain the temporary AES key (and consequently also the keys granted to a new device during the joining phase).

4.1 ASM Model of the *Joining Procedure*

As shown in the excerpt of Code 1, the controller and the slave participating to the *joining procedure* are modeled as two agents with their programs running independently. These programs can be viewed as two separate finite state machines (FSMs) that change their internal state only if a message arrives from the other agent, or when a timer has expired.

In Fig. 3, the flow chart of the corresponding ASM model is depicted. There are two swim-lines, one for node A, the controller, and one for node B, the slave. The node internal states are coloured in light green. The flow of a node behaviour is vertical, whereas the horizontal flow is given by the messages exchanged between the two nodes. The mapping from Z-Wave protocol description to ASMs is rather straightforward: for each message exchange in the green box of Fig. 2, there is a corresponding *state* in Fig. 3 and a *transition rule* is defined. The INPUT box on some states, e.g., on *INSERT PIN* state, models the fact that a user is asked to perform an action without which it is impossible to proceed to the next state. Mimicking the protocol requirements, every time a control fails (e.g, in case there is a security class mismatch between the requested classes and the granted classes, or the wrong ECDH curve is asked - at the moment the only possible curve is ECDH on Curve25519), the node sends an error message to the other node and terminates its execution.

As explained in Section 3.2, an ASM state consists of objects' domains and functions defined on them. For the sake of brevity, we cannot list all the functions we defined. We discuss the most important ones. The internal state of agents is a controlled function (*slaveState* or *controllerState*) that associates a slave or a controller to its actual state. Protocol messages are also represented by a controlled function named *protocolMessage*, where the first parameter represents the message sender, while the second indicates the receiver. As another example, each agent has a matching type. The slave type *SLAVETYPE* is used to specify the device type and the security class the device belongs to. A monitored function *slave: SLAVE → SLAVETYPE* is invoked during the agent initialisation phase to setup the device type. Under the assumption of joining a native S2 Security device, as explained in Section 3.1, the input of the PIN code takes place on the controller side in the *INSERT PIN* state, otherwise, the user inputs the PIN code on the slave side during the *INSERT PIN CSA* state. For the slave, the invocation of the *slave* function corresponds to the NIF of Fig. 2 sent by the slave to the controller before the *joining procedure* starts.

Both in the controller and slave models, most of the transition rules have the same structure: activation is demanded to a rule guard that controls the current state, the message received, and the time left. The core of a rule consists of function updates changing the internal state of the agent, setting a timer, and crafting the next messages expected by the Z-Wave protocol.

As an example, we describe in depth the *KexReport* rule in Code 2, the other rules are similar:

- The first line of code selects the controller or the slave with whom the agent is communicating (`choose $ctrl in Controller`) and emulates the *NodeID* that is used in the network to identify a device uniquely.
- Lines 2-4 specify the guard, which activates the updates if it is true. The guard checks: (*i*) the internal state of the slave (in this case the slave must be in *LEARN_MODE* state), (*ii*) the message received (in this case the message must be part of the negotiation part of the protocol, i.e., *KEX_GET*), and (*iii*) if there is still some time for the execution of the rule (by means of the mon-

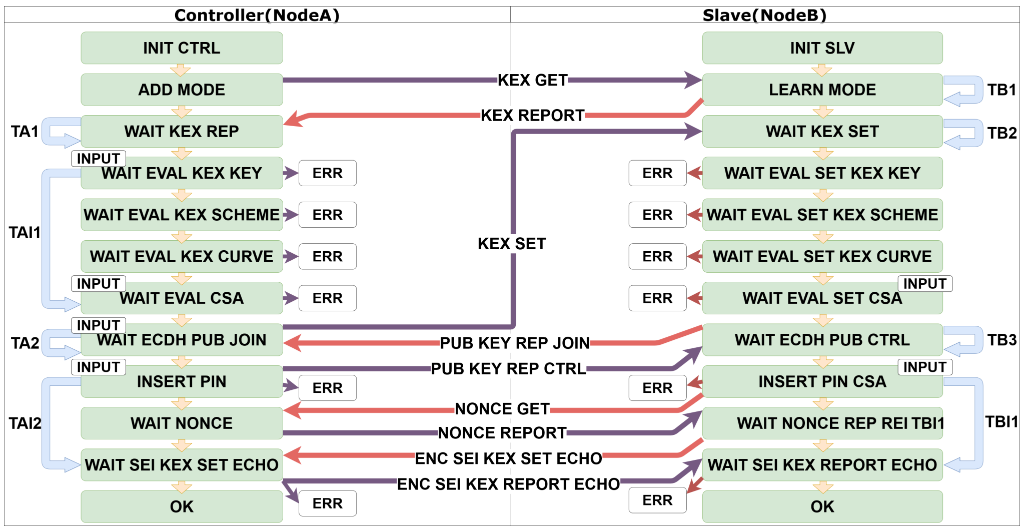

Figure 3: ASM model of Z-wave *joining procedure*.

itored function *passed* which tells if a timer has expired).

- On Lines 6-8, the slave: (*i*) updates its internal state to *WAIT_KEX_SET*, (*ii*) activates timer *TB2*, and (*iii*) deactivates timer *TB1*.
- On Lines 9-17, the slave sends the *KEX_REPORT* message to the correct controller (line 9), and with the expected payload (lines 11-17).
- On lines 18-25, the slave stores the message payload locally for further computation (thanks to the syntax `(self, self)`).

Some other transition rules are not covered by a timer, or they execute only controls over the payload of an already received message (e.g., *EvalGrantKexKey* rule checks if the intersection between *REQUESTED KEY* and *GRANTED KEY* is not empty, and then it saves the *GRANTED KEY* locally).

Model Validation. Our model of the *joining procedure* has been validated by using the `AsmetaV` tool in combination with the simulator `AsmetaS`. We expressed one scenario for each requirement in the Z-Wave protocol documentation. Scenarios are written in the `Avalla` language, providing special commands to: **set** the values of monitored functions, perform one **step** of simulation, **exec** rules or function updates, **check** that some properties hold. By playing with a combination of these commands, we were able to express significant scenarios and check that the model behaves as expected.
Code 3 shows (a fragment of) the validation scenario of the security class mismatch case. In this scenario, the *joining procedure* terminates throwing the error

```
1   choose ctrl in Controller with true do
2       if(slaveState( self ) = LEARN_MODE
3       and protocolMessage( ctrl , self ) = KEX_GET
4       and not passed( TB1 )) then
5           par
6               slaveState( self ) := WAIT_KEX_SET
7               startTimer( TB2 ) := true
8               startTimer( TB1 ) := false
9               protocolMessage( self, ctrl ) := KEX_REPORT
10              // KEX REPORT payload
11              reqCsa( self , ctrl ) := slvCsa( self )
12              reqSkex( self , ctrl ) := slvSkex( self )
13              reqEcdh( self , ctrl ) := slvEcdh( self )
14              reqS2Access( self , ctrl ) := slvS2Access( self )
15              reqS2Auth( self , ctrl ) := slvS2Auth( self )
16              reqS2Unauth( self , ctrl ) := slvS2Unauth( self )
17              reqS0( self , ctrl ) := slvS0( self )
18              //save for control of authenticity
19              reqCsa( self, self ) := slvCsa( self )
20              reqSkex( self, self ) := slvSkex( self )
21              reqEcdh( self, self ) := slvEcdh( self )
22              reqS2Access( self, self ) := slvS2Access( self )
23              reqS2Auth( self, self ) := slvS2Auth( self )
24              reqS2Unauth( self, self) := slvS2Unauth( self )
25              reqS0( self, self ) := slvS0( self )
26          endpar
27      endif
```

Code 2: KexReport rule.

KEX_FAIL_KEX_KEY. On line 4-5, we check that the scenario is correctly set at the beginning of the *joining procedure*. Then, on line 6-7, we set the type of the controller and of the slave (recall that the slave is a native S2 security smart light). On lines 13-14, we simulate the user disabling class S0 security and an interference modifying the payload of *KEX_REPORT*, i.e., the corruption of S2 Unauthenticated security class.

After some intermediate simulation steps and checks, on lines 18-20, we check that the controller (i) sends the *KEX_FAIL_KEX_KEY* error message to the slave, (ii) goes in the *ERROR_C* error state, and (iii) checks if the slave is still in a *WAIT_KEX_SET* state. At the end, on lines 23-24, we check that, after receiving the *ERROR_S* error message from the controller, the slave has gone into the *ERROR_S* error state, as expected.

```
1   scenario validation_protocol_ctrl_key_mismatch
2   load zwaveProtocol_join.asm
3
4   check slaveState(nodeB) = INIT_SLV
5   and controllerState(nodeA) = INIT_CTRL;
6   set controller(nodeA) := CONTROLLER_S2;
7   set slave(nodeB) := LIGHT_NAT;
8   ...
9   step
10  ...
11  exec
12    par
13      ctrlS0(nodeA) := false
14      slvS2Unauth(nodeB) := false
15    endpar;
16  ...
17  step
18  check protocolMessage(nodeA,nodeB)= KEX_FAIL_KEX_KEY
19  and slaveState(nodeB) = WAIT_KEX_SET and
20  controllerState(nodeA) = ERROR_C;
21  ...
22  step
23  check protocolMessage(nodeA,nodeB)= KEX_FAIL_KEX_KEY and
24  slaveState(nodeB) = ERROR_S and
25  controllerState(nodeA) = ERROR_C;
```

Code 3: Validation of key mismatch scenario.

Property Verification. In order to verify that a specific security property holds in the model, the property must be expressed as a CTL (Computation Tree Logic, CTL for short) temporal logic formula and proved to be true. We used the `AsmetaSMV` tool that checks if a CTL formula holds in an ASM model by exploiting the `NuSMV` model checker. The result, if the model checker terminates, is a boolean condition: if it is `False`, it means that the property is violated at least once (and the tool returns the violating path), in case of `True`, the property holds.

We examined many reachability and safety properties, and none of the properties we tested was ever violated. For example, in the security class negotiation phase of the *joining procedure*, we checked:

- If there exists a state in the future in which the *CONTROLLER_S2* (*nodeA*) sends *KEX_SET* with an illegal scheme, expressed with the CTL formula:

  $$\neg EF(\text{controllerState(nodeA) = ERROR\_C} \wedge \text{slaveState(nodeB) = ERROR\_S} \wedge \text{protocolMessage(nodeA,nodeB) = KEX\_FAIL\_KEX\_KEY})$$

  The evaluation of the formula is `True`, since without an external interference the *CONTROLLER* always sends the correct scheme.
- There is not a state in the future in which a *SLAVE* sends *KEX REPORT* with an incompatible curve.

  $$\neg EF(\text{controllerState(nodeA) = ERROR\_C} \wedge \text{slaveState(nodeB) = ERROR\_S} \wedge \text{protocolMessage(nodeA,nodeB) = KEX\_FAIL\_KEX\_CURVE})$$

  The evaluation of the formula is `True`, since without an external intervention it is impossible the slave sends an incorrect curve.

We also specified CTL formulas to check the correctness of the parts of the protocol relying on timers. In particular, we checked that a timer is running only on the states it is assigned to. For example, for timer *TA1*, the formula is:

$$\neg AG((\text{controllerState(nodeA) != WAIT\_KEX\_REP} \wedge \text{controllerState(nodeA) != TIMEOUT\_C}) \Rightarrow \text{startTimer(TA1)= false})$$

In this formula, we checked that the only states in which timer *TA1* might be active are either $WAIT\_KEX\_REP$ or $TIMEOUT\_C$.

All the CTLs tested ensure that the *joining procedure* behaves correctly.

4.2 ASM Model with an Attacker

In this section, we describe how we extended the *joining procedure* model presented in Section 4.1 with two new agents, *ESLAVE* and *ECONTROLLER*, modelling an evil adversary trying to subvert the protocol. *ECONTROLLER* models an attacker impersonating a legitimate controller when communicating with the slave, while *ESLAVE* does the opposite. The two agents model an active attacker controlling the network, thus able to intercept, modify, delete and inject any message. In particular, we gave the attacker the ability to execute a brute force attack to retrieve the PIN code. This specific capability was given to the attacker since, while formalising the protocol, we noticed that the duration of timers *TA1* and *TA2* might be overestimated, leaving enough time to an intruder to execute a brute force attack to retrieve the PIN code. Our goal is then to verify if a MITM can break the *joining procedure*, obtaining by the entitled controller the granted keys associated to the legitimate slave security classes.

In short, the attack works as follows: the attacker exploits the session of the protocol between *ECONTROLLER* and *SLAVE* (called *EC-S* for short) in order to obtain from the legitimate slave the first 5 digits of its public key, then (s)he uses them in the session between *CONTROLLER* and *ESLAVE* (called *C-ES* for short) to obtain the keys in place of the legiti-

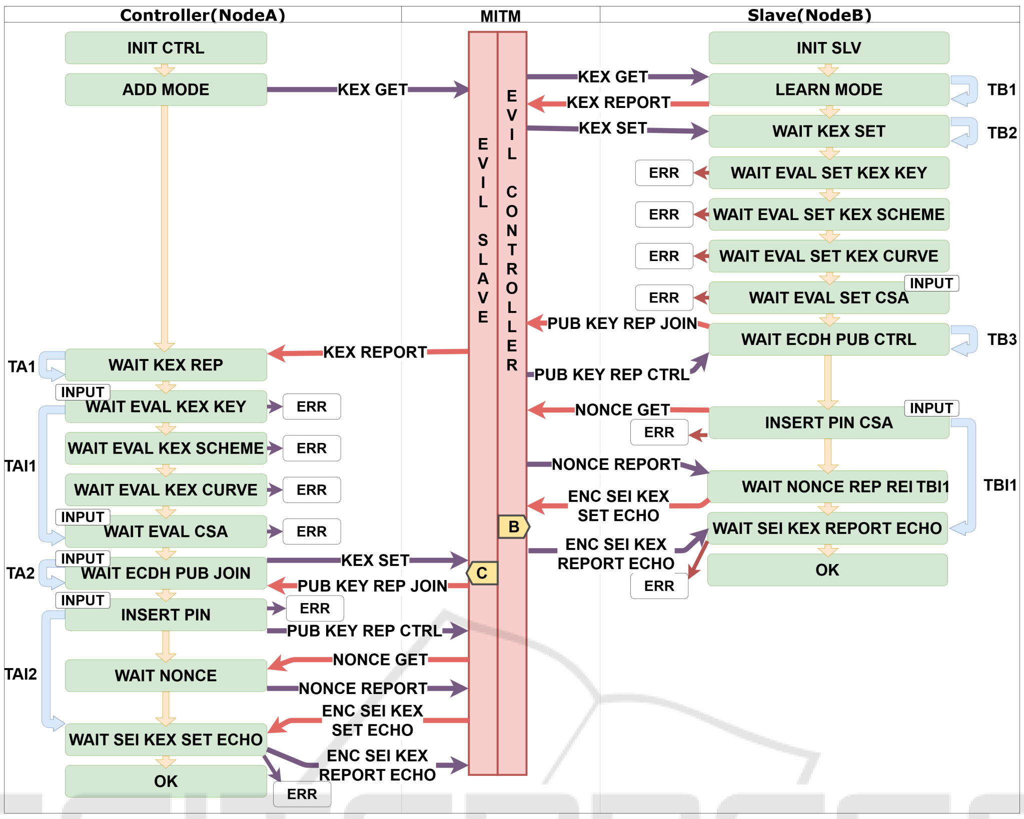

Figure 4: Flow chart of the MITM attack.

mate slave. To this aim, the ASM model works with evil slave and evil controller sharing a memory, where they store all the messages exchanged in the two sessions of the protocol. In Fig. 4, the flow chart of the ASM model corresponding to the whole MITM attack is depicted (in the picture, the brute force attack to obtain the PIN is labeled with B, whereas label C identifies the attack on the public key using generated by knowing the PIN). In this case, there are three swimlines: the one on the left for node A, the legitimate controller, the one on the right for node B, the legitimate slave, and the one in the center for the two ASM agents emulating the attacker. The right part of the flow chart depicts the messages exchanged during the *EC-S* session of the protocol, while the left part shows the *C-ES* session.

The attack begins after the new device has started the *joining procedure* by announcing its device type with the NIF, with the attacker impersonating the controller and sending the *KEX GET* message to the device (i.e., the *EC-S* session starts). The attacker can continue the execution of the protocol until (s)he receives the information of her/his interest, i.e., message *ENC SEI KEX SET ECHO*, which is the target of the first part of the attack, since it can be used to retrieve the initial 5 digits of the slave public key. The 5 digits are needed both in the *EC-S* and in the *C-ES* session of the protocol. In the first case, they are needed to complete the *EC-S* session without making the user suspect that the device has failed the join. In the second case, they are necessary to continue the *C-ES* protocol session (recall that, according to the *joining procedure*: first, the slave sends to the controller its public key with the first 5 digits obfuscated; then, during the OOB authentication, the user manually inserts the first 5 digits; finally, the controller uses the slave public key and its own private key to generate with ECDH the temporary key K_t used to encrypt the requested keys). In this case, there is a time constraint: the attacker must get the 5 digits in the *EC-S* protocol session in order to be able to send to the legitimate controller a public key with the correct first 5 digits (the ones the user will insert during OOB authentication) before *TA2* runs out in the *C-ES* session.

In order to obtain the slave public key in the *EC-S* protocol session, the attacker has two options (before *TA1* and *TA2* run out):

1. To perform a brute-force attack on temporary AES 128 bits key K_t, then derive the slave public key by knowing K_t and the controller private

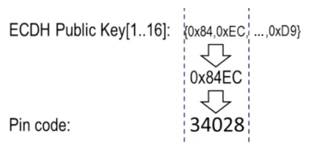

Figure 5: PIN number of bytes.

key (as described in Section 3.1, K_t is generated by the (evil) controller by using her/his own private key and the slave's public key). A brute force attack on a 128 bits key would need to try 2^{127} keys (on average), thus the time required would be a lot longer than *TA1+TA2*. Therefore, this option is not feasible.

2. To take advantage of the fact that K_t can be generated using ECDH either by knowing the legitimate slave's private key and the evil controller's public key, or the evil controller's private key and the legitimate slave's public key. The attacker (acting as a controller) knows her/his own private key and the legitimate slave's public key a part from the first 5 digits. According to Z-Wave documentation (see Fig. 5), the PIN code must be 5 decimal digits representing the value of the first 2 bytes of the node's device specific key. Since the slave public key has the first 2 bytes obfuscated, the evil controller has to guess only 2 bytes in order to find the correct public key. Therefore, the size of the search space of the key is reduced from 2^{128} to 2^{16}. More specifically, the attacker has to generate with ECDH a key K for all 2^{16} combinations, and then check for which combination it yields $K = K_t$ (by decrypting the 6 bytes message *ENC SEI KEX SET ECHO*). By using the OpenSSL command (version 1.0.2j) in decrypt mode on a notebook with 8 Gb of RAM and Intel i7-8550U CPU (see Table 2 for performance information), considering a 16 bytes packet, the slave public key can be retrieved in around 65 ms. Therefore, *TA2* timeout (10s) gives the evil controller plenty of time to complete the attack.

Table 2: AES 128 CCM decrypt single core performance.

	16 bytes	64 bytes
Number of Packets in 3 sec	304057417	184699602

In the *EC-S* parallel session of the protocol, at the beginning, the attacker (impersonating the slave) is in the *ADD MODE* state and intercepts the *KEX GET* message. Although the attacker can respond immediately, (s)he waits to send the *KEX REPORT* message received from the slave in order to take time. However, (s)he must send it before timer *TA1* expires. Afterwards, the legitimate controller completes the negotiation phase and reaches the point in the protocol where the joining device has to insert its public key (with the first 5 digits obfuscated) before *TA2* expires. At this point the attacker could:

- Use the public key (s)he obtained in the *EC-S* session, but (s)he does not know the associated private key and it would take too long to generate it with another brute force attack.
- Create a pair of public-private keys and send the public one to the controller. However, if the first 5 digits do not match with the ones obtained in the *EC-S*, the OOB authentication would fail.
- Create the public-private key pair so that the public key contains the first 5 digits obtained in the *EC-S* session.

The last option slightly differs from the first one: in the latter case, the public key is crafted in a way that only the first 5 digits (i.e., 2 bytes) correspond to the ones of the legitimate slave public key, not the whole key. This fact simplifies the attack: in this case, the attacker may create in advance at least one key pair (public and private) of the 2^{16} possible ones (i.e., the key pairs with the public key differing on the first 2 bytes). Thus, the feasibility of the attack depends on the number of keys the attacker has to generate to be confident, with a good probability, that (s)he has got at least one copy of the keys (s)he will need to complete *EC-S* session of the protocol.

This probability can be expressed, as usual, as the ratio between successful outcomes and all possible outcomes. In our case, it yields:

$$P(gen\_success) = \frac{\binom{y-2^{16}+2^{16}-1}{2^{16}-1}}{\binom{y+2^{16}-1}{2^{16}}}$$

Being y the total number of keys to generate, the numerator corresponds to the multi-set binomial coefficient, which represents the number of ways of selecting a total of 2^{16} out of y - 2^{16} keys with unrestricted repetition. The denominator shows all the possible combination of the 2^{16} unique public keys over a total of y generated keys.

We use the formula to find a feasible y, i.e., the number of keys the attacker needs to generate off-line to be sure with $P(gen\_success)$ probability of finding the 2^{16} unique keys (s)he will need. With $y = 2^{34}$, we obtain $P(gen\_success) = 80\%$. In average, for a good home computer, the time interval for the generation of a single key pair is between 10^{-4} and 10^{-6} seconds.

During the protocol attack, the evil slave looks for a key with the same first 2 bytes of the slave public key obtained in the first part of the attack among the ones in her/his hash table of pre-computed keys (see Table 3 for an example), and sends it to the controller.

Table 3: Example of crafted public keys.

00000	26467 58443 24926 55534 27025 55113 08637
00001	43605 13128 01516 47089 25370 27578 61791
...	...
65365	49316 10780 35451 55217 51086 38075 23415

In a ideal run of the protocol, a user should check the rest of the key digits visually, but the lack of attention of the user may make the attack fully successful.

Ironically, older devices that have been updated to S2 Security are resistant to the attack. In fact, in this case, the PIN code required is 4 bytes long (not only 2 bytes). Thus, the brute force attack would require longer than *TA1 + TA2*.

Model Validation. The validation phase is slightly different from 4.1 because the the attacker does not propagate the error messages that are described in the protocol. In this way, the complexity of the MITM model is reduced and all the validation scenarios executed for the *joining procedure* are not of interest in this analysis. The message flow remains unchanged, so we do not need to revalidate those parts. The validation of this model is achieved by reproducing the path obtained in the verification phase that confirms the MITM attack feasibility. The scenario is represented in Code 4.

The *nearToEnd* monitored function acts as an internal timer that computes on the MITM agents an approximation of the CONTROLLER timer. This monitored function is essential to maintain the model as realistic as possible. In fact, in a real environment, the attacker can only guess when the timer is started when (s)he has received the messages. We added to the MITM attacker model an invariant that grants the temporal consistency between the *passed* and the *nearToEnd* monitored function.

Property Verification. The verification is easy for the MITM model, and it consists of the following CTL formula:

- There is not a state in the future in which the *CONTROLLER* and the *SLAVE* are both in the OK state, that represents the final state they reach only when the protocol runs as expected.

$\neg EF(\text{controllerState(nodeA)} = \text{OK\_C} \wedge \text{slaveState(nodeB)} = \text{OK\_S})$

As expected, the model checker returns a path leading to a state where the property does not hold, meaning that an attack can be successful.

```
scenario validation_protocol_mitm
load zwaveProtocol_join.asm

check slaveState(nodeB) = INIT_SLV and
      controllerState(nodeA) = INIT_CTRL;
set controller(nodeA) := CONTROLLER_S2;
set slave(nodeB) := GARAGE_LOCK_NAT;
set passed(TA1) := false;
...
step
...
set nearToEnd(TA1) := true;
step
check protocolMessage(nodeES,nodeA)=KEX_REPORT and
protocolMessage(nodeEC,nodeB) = PUB_KEY_REP_CTRL and
slaveState(nodeB) = WAIT_ECDH_PUB_CTRL and
controllerState(nodeA) = WAIT_KEX_REP;
...
step
...
check protocolMessage(nodeA,nodeES)=EC_KEX_REPORT_ECHO
and protocolMessage(nodeEC,nodeB)=EC_KEX_REPORT_ECHO and
slaveState(nodeB) = OK_S and controllerState(nodeA) = OK_C;
```

Code 4: Successful MITM attack scenario.

5 CONCLUSION

In this paper, we present a formal analysis of the Z-Wave protocol using S2 Security class, a security layer required in every new certified IoT device. In particular, we focus on the *joining procedure*, which is the core part of the S2 Security layer, allowing a new device to join the Z-Wave net and to obtain the symmetric key to be used for future communication.

We model the protocol requirements by means of an ASM model. As a consequence, we are able to provide a rigorous and precise protocol description, which is often ambiguous and fragmented in the official documentation. Moreover, we validate the protocol execution by means of a scenario-based technique, and we prove a sequence of reachability and safety temporal properties to guarantee the correctness of the *joining procedure*. We also formalise the behaviour of an external actor behaving maliciously with the capability of intercepting, modifying, deleting and injecting messages. In particular, we give the attacker the ability to execute a brute force attack to retrieve the PIN code.

The verification process revealed the possibility of executing a MITM attack for the PIN-based OOB authentication method under certain time constraints. Its practical feasibility was confirmed by the Silicon labs and Z-Wave Alliance.

Our results show the importance of exploiting formal methods during the design process of a security protocol in order to find vulnerabilities since the early stages of protocol development.

As a short-term objective, we plan to further investigate the Z-Wave protocol, both by modeling the application layer, and by covering the whole S2 Security layer, searching for other vulnerabilities.

Since the mathematical base of formal methods discourages designers from their usage, as a long-term objective, we plan to develop a "user-friendly" environment for the formal specification, verification and development of IoT security protocols.

REFERENCES

Arcaini, P., Gargantini, A., and Riccobene, E. (2017). Rigorous development process of a safety-critical system: from ASM models to Java code. *Int. J. Softw. Tools Technol. Transf.*, 19(2):247–269.

Arcaini, P., Gargantini, A., Riccobene, E., and Scandurra, P. (2011). A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*, 41(2):155–166.

Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuellar, J., Drielsma, P. H., Heám, P. C., Kouchnarenko, O., Mantovani, J., Mödersheim, S., von Oheimb, D., Rusinowitch, M., Santiago, J., Turuani, M., Viganò, L., and Vigneron, L. (2005). The AVISPA tool for the automated validation of internet security protocols and applications. In *Computer Aided Verification*, pages 281–285. Springer.

Badenhop, C. W., Graham, S. R., Ramsey, B. W., Mullins, B. E., and Mailloux, L. O. (2017). The z-wave routing protocol and its security implications. *Computers & Security*, 68:112–129.

Basin, D., Dreier, J., Hirschi, L., Radomirovic, S., Sasse, R., and Stettler, V. (2018). A formal analysis of 5g authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1383–1396, New York, NY, USA. Association for Computing Machinery.

Bonfanti, S., Gargantini, A., and Mashkoor, A. (2020). Design and validation of a C++ code generator from abstract state machines specifications. *J. Softw. Evol. Process.*, 32(2).

Börger, E. and Raschke, A. (2018). *Modeling Companion for Software Practitioners*. Springer.

Börger, E. and Stärk, R. (2003). *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag.

Dolev, D. and Yao, A. (1983). On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208.

Eldefrawy, M., Butun, I., Pereira, N., and Gidlund, M. (2019). Formal security analysis of lorawan. *Computer Networks*, 148:328 – 339.

Fouladi, B. and Ghanoun, S. (2013). Security evaluation of the z-wave wireless protocol. In *Proceedings of Black Hat USA*, pages 1–2.

Hofer-Schmitz, K. and Stojanović, B. (2020). Towards formal verification of IoT protocols: A review. *Computer Networks*, 174:107233.

Kim, K., Cho, K., Lim, J., Jung, Y., Sung, M., Kim, S., and Kim, H. (2020). What's your protocol: Vulnerabilities and security threats related to z-wave protocol. *Pervasive and Mobile Computing*, 66.

Melaragno, A. P., Bandara, D., Wijesekera, D., and Michael, J. B. (2012). Securing the zigbee protocol in the smart grid. *Computer*, 45(4):92–94.

Mohsin, M., Sardar, M. U., Hasan, O., and Anwar, Z. (2017). Iotriskanalyzer: A probabilistic model checking based framework for formal risk analytics of the internet of things. *IEEE Access*, 5:5494–5505.

Qiu, Y. and Ma, M. (2016). A pmipv6-based secured mobility scheme for 6lowpan. In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6.

Qiu, Y. and Ma, M. (2018). Secure group mobility support for 6lowpan networks. *IEEE Internet of Things Journal*, 5(2):1131–1141.

Sun, D.-Z. and Sun, L. (2019). On secure simple pairing in bluetooth standard v5.0-part i: Authenticated link key security and its home automation and entertainment applications. *Sensors (Basel, Switzerland)*, 19(5):1158.

Unwala, I., Taqvi, Z., and Lu, J. (2018). Iot security: Zwave and thread. In *2018 IEEE Green Technologies Conference (GreenTech)*, pages 176–182.

Viganò, L. (2006). Automated security protocol analysis with the avispa tool. *Electronic Notes in Theoretical Computer Science*, 155:61 – 86. Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI).

Z-wave (2020). Z-wave protocol. https://www.z-wave.com/.

Z-Wave (2020). Z-wave protocol. https://www.silabs.com /documents/login/miscellaneous/SDS13783-Z-Wave-Transport-Encapsulation-Command-Class-Specification.pdf.