7-2017

The Z-Wave Routing Protocol and Its Security Implications

Christopher W. Badenhop

Scott R. Graham
Air Force Institute of Technology

Benjamin W. Ramsey
Air Force Institute of Technology

Barry E. Mullins
Air Force Institute of Technology

Logan O. Mailloux
Air Force Institute of Technology

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

The Z-Wave routing protocol and its security implications

Christopher W. Badenhop [a,*], Scott R. Graham [a], Benjamin W. Ramsey [a], Barry E. Mullins [a], Logan O. Mailloux [b]

[a] Department of Electrical and Computer Engineering, Air Force Institute of Technology, WPAFB, USA
[b] Department of Systems Engineering and Management, Air Force Institute of Technology, WPAFB, USA

ARTICLE INFO

Article history:
Received 26 October 2016
Received in revised form 28 February 2017
Accepted 7 April 2017
Available online 12 April 2017

Keywords:
Z-Wave
Network security
Home automation security
Routing protocol analysis
Reverse engineering
Embedded systems security

ABSTRACT

Z-Wave is a proprietary technology used to integrate sensors and actuators over RF and perform smart home and office automation services. Lacking implementation details, consumers are under-informed on the security aptitude of their installed distributed sensing and actuating systems. While the Physical (PHY) and Medium Access Control (MAC) layers of the protocol have been made public, details regarding the network layer are not available for analysis. Using a real-world Z-Wave network, the frame forwarding and topology management aspects of the Z-Wave routing protocol are reverse engineered. A security analysis is also performed on the network under study to identify source and data integrity vulnerabilities of the routing protocol. It is discovered that the topology and routes may be modified by an outsider through the exploitation of the blind trust inherent to the routing nodes of the network. A Black Hole attack is conducted on a real-world Z-Wave network to demonstrate a well-known routing attack that exploits the exposed vulnerabilities. As a result of the discoveries, several recommendations are made to enhance the security of the routing protocol.

1. Introduction

Z-Wave is an implementation of a complete Internet of Things (IoT) substrate, containing well-defined communication, networking, and application layer protocols. The user composes Z-Wave capable sensors, actuators, controllers, routers, and Internet gateways to provide home and office automation services. Example automation services include security monitoring, smart power management, and climate control; still, the device-level granularity of system composition allows the user to realize more exotic IoT services. The communication substrate uses a proprietary radio stack, where the PHY layer and MAC layer are defined in ITU G.9959 (2012). Additionally, a significant portion of the application layer is revealed in OpenZwave source code (OpenZwave, 2016). Sitting between the MAC and application layer is the optional network layer, which handles multihop routing.

While the specifications of the PHY, MAC, and a portion of the application layer of the Z-Wave protocol stack are publicly accessible, few details regarding the network layer exist within the public domain. The most significant reference is found in Paetz (2013), where the author indicates that Z-Wave uses *static source routing*. Moreover, routes are calculated from a centralized routing table and embedded into routed messages to dictate their forwarding behavior (Paetz, 2013). In Fuller et al. (2017), the basic Source Route (SR) forwarding mechanism for Z-Wave is described. Beyond these sources, several

[*] *Corresponding author.*
E-mail address: cwbadenhop@gmail.com (C.W. Badenhop).

http://dx.doi.org/10.1016/j.cose.2017.04.004

non-peer-reviewed blogs are available that provide supporting details on the protocol (Zwave Protocol 1, 2016; Zwave Protocol 2, 2016). While they correlate with Paetz (2013), they do not provide new information about the routing protocol. To date, an open source implementation of Z-Wave routing does not exist. While OpenZwave is an open source implementation of a Z-Wave controller for a PC, the routing logic resides in the firmware of the required USB Z-Wave transceiver dongle.

The contributions of the work herein are as follows. First, a significant portion of the Z-Wave network routing protocol is reverse engineered to include a detailed understanding of the frame forwarding and topology management mechanisms. Second, the aspects of these mechanisms are analyzed to identify source and data integrity vulnerabilities of the protocol. The vulnerabilities are exploited to conduct a Black Hole attack on a real-world Z-Wave network. Several recommendations are provided to enhance the security of the routing protocol based on the vulnerabilities discovered on the network under study.

The remainder of this paper is as follows. Section 2 provides a brief description of the salient aspects of Z-Wave and related work that is relevant to the paper. Section 3 describes the data acquisition, exploitation, and analysis methodology used for reverse engineering and performing a security assessment of the protocol. The reverse engineering results are composed in Section 4 to provide the most complete and publicly available model of the Z-Wave routing protocol. Section 5 provides the results of the security assessment, including existing security mechanisms offering some resistance to source and data integrity attacks. In the presence of the existing security mechanisms, a Black Hole attack is conducted to demonstrate the synthesis of discovered vulnerabilities on a Z-Wave network. Several recommendations to the protocol are included in this section. This is followed by Section 6, which provides considerations for future work and conclusions.

2. Z-Wave background

Using the Z-Wave physical layer, messages are asynchronously exchanged over the RF medium as MAC Protocol Data Unit (MPDU) frames. A MPDU contains a header, consisting of identification and control fields. The MPDU payload contains data pertaining to an application layer command, query, or report. Optional layers exist between the MPDU header and application layer, including the network and security layers. Several other aspects of Z-Wave are provided below.

2.1. *Pairing operation*

A distinguishing aspect of Z-Wave is the manner in which devices enter and leave the network. This *inclusion* or *pairing* process is similar to Bluetooth device pairing. A user wishing to add a device to the network first puts the Z-Wave controller and the new device into a pairing mode. While placing a device in pairing mode is device specific, this is commonly achieved by pressing a button or physically resetting the device. While in the pairing mode, the controller adds any device found to also be in pairing mode. Depending on the controller implementation, it may present a list of discovered devices and allow the user to select ones to add. Alternatively, the controller may add all discovered devices within physical proximity without discrimination. Examples of each type of controller include the Mi Casa Verde Vera (Vera, 2016) and Aeon Labs Z-Stick (Z-Stick, 2016), respectively. Removing a device from a Z-Wave network is accomplished in a similar manner.

2.2. *Node identification*

Z-Wave devices are identified by a 4-byte *home ID*, which is assigned by the controller to a device during the pairing process. All nodes paired to a given controller share the same home ID, which is assigned to the controller by the vendor in the factory.

The second form of identification is the *node ID*. A node ID is a byte value also assigned by the controller to a device during the pairing process. The controller node always has the lowest node ID of 1. The first device paired to a controller has a node ID of 2. Subsequent pairing operations result in the assignment of an unused and monotonically increasing node ID.

2.3. *Command class*

Z-Wave networks are composed of controllers, sensors, and actuator devices. To ensure device compatibility, the application layer protocol is well-defined (OpenZwave, 2016); however, not all devices require the ability to participate in every application layer transaction. For example, a light switch does not need to know how to respond to a request for a temperature reading. Consequently, the application layer is partitioned by functionality into a series of *command classes*. Within each command class is a subset of the application layer commands pertaining to the class. A given Z-Wave device belongs to one or more command classes and the associated application layer protocol functionality is included during compilation of its firmware image. A device announces this set of supported classes to the controller during the pairing operation. Each command class is a unique byte value (OpenZwave, 2016). For example, a light switch may announce that it uses the *Binary Switch* command class, which provides commands to get, set, and report the state of the switch (OpenZwave, 2016). Given the announcement, the controller is now aware of a subset of commands that the light switch obeys. Commands sent to the light switch from unsupported command classes are ignored.

2.4. *Related work*

While there are publications regarding the reverse engineering and security assessment of other aspects of Z-Wave systems, the work herein is the first to examine security vulnerabilities of the routing protocol. In Badenhop et al. (2016b), the non-volatile memory components associated with a common Z-Wave transceiver chip are extracted and analyzed. Several data structures, including the node adjacency table used for route construction by a Z-Wave controller, are identified. Building on the results from Badenhop et al. (2016b), the authors analyze the Z-Wave security layer in Badenhop and Ramsey (2016). They discover the relationship between the

Advanced Encryption Standard (AES) keys and where they are stored in the Electronically Erasable/Programmable Read-Only Memory (EEPROM) of the device. This leads to the identification of a key extraction vulnerability, where the keys are extracted from devices lacking physical security and used to send encrypted and authenticated commands to other secure devices in the network.

Other related activities include one of the earliest investigations of Z-Wave security, where a vulnerability is exploited in a Z-Wave door lock device which allows attackers to reset the Z-Wave communication key. With the key reset and known by the attacker, the door can be disengaged remotely (Fouladi and Ghanoun, 2013). In Fuller and Ramsey (2015), a Z-Wave controller is exploited, using its Internet access point and web server, to add arbitrary rogue devices to a home automation network. An intrusion detection system is proposed for Z-Wave in Fuller et al. (2017) to detect outsider activities within the network. The application protocol is exploited in Hall and Ramsey (2016) to lower the lifetime of fluorescent bulbs powered through Z-Wave power switches. The authors find that rapidly toggling the power to these devices using the Z-Wave application protocol reduces operational lifetime of the bulbs. The susceptibility of Z-Wave networks to passive and active reconnaissance, device enumeration, and fingerprinting are explored in Badenhop et al. (2015), Patel and Ramsey (2015), Bihl et al. (2015), and Hall et al. (2016).

3. Reversing engineering and security assessment methodology

Reverse engineering and security assessment are performed using black box analysis (Stutton et al., 2007). The effort consists of a series of experiments, where the network under study is exposed to stimuli while passively collecting frames emitted by the devices of the network. The captured frames are analyzed to reach conclusions that motivate further experimentation. Stimuli are generated by invoking legitimate network traffic using the devices of the network, which is especially useful for protocol reverse engineering. Alternatively, a *Software Defined Radio (SDR) test platform* is used to interact with devices in the network under study at the network layer. The test platform has the capability to send and receive Z-Wave frames, allowing the edge-cases of the protocol to be examined to discover security vulnerabilities.

3.1. *Network under study*

The network under study is a collection of four Z-Wave devices and a single controller, sharing a Z-Wave home ID of 0x018509FF. An Aeon Z-Stick2 is the Z-Wave controller for the network. It runs the static controller library version 2.78 and is Node 1. Nodes 3, 4, and 5 are Aeon appliance power switches. The devices report using Z-Wave library version 3 and protocol version 2.78. To avoid a network composed of devices from a single vendor and firmware version, Node 2 is a GE outdoor power switch with Z-Wave library version 6 and protocol version 3.67. The four devices are always powered, so they do not require *beaming* to wake them up to receive frames (ITU G.9959, 2012). Each has the capability of routing Z-Wave frames and are depicted in Fig. 1.

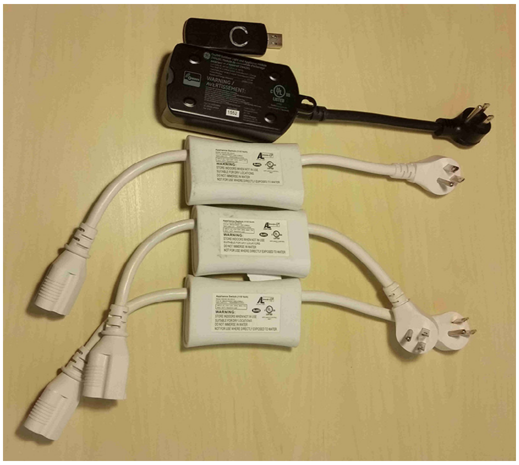


Fig. 1 – Devices used in the network under study. From top to bottom, this includes an Aeon USB ZStick2, a GE Outdoor Switch, and three Aeon Appliance Switches.

While several topologies are examined for the work herein, a single topology is used to provide a consistent demonstration platform to present the results of the reverse engineering and security analysis efforts. The demonstration topology of the network under study is shown in Fig. 2. All devices in the topology are assumed to transmit at approximately the same power level. This allows the bi-directional link assumption to be made for all links of the topology (Narayanaswamy et al., 2002). The power switches are placed in a ring configuration, and the controller is only connected to Node 2. The topology is chosen to provide multiple paths between nodes and to force multihop routing between the controller and Nodes 3, 4, and 5.

To enable network-wide passive collection, the devices of the demonstration topology are physically located within a single hop distance; however, this results in a fully-connected mesh topology. The demonstration topology is derived from the mesh topology using the security vulnerabilities identified in Section 5. Links are manually removed from the fully connected topology using Neighbor List (NL) and SR cache update messages until the topology conforms to Fig. 2. Being all within transmission range, the nodes in the network under study observe shorter routes; however, they do not update their

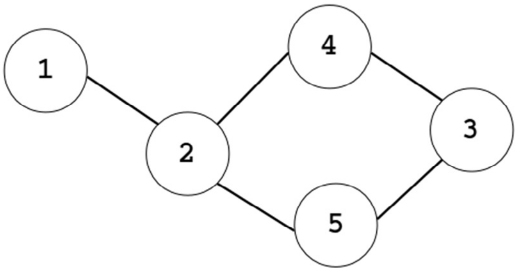


Fig. 2 – The topology of the network under study used for demonstrating the results of the reverse engineering and security analysis findings.

topology information with more optimal routes. As a result, the network topology remains stable.

3.2. The SDR test platform

From ITU G.9959 (2012), there are three data rate configurations R1, R2, and R3. Each configuration provides a unique combination of symbol rate, center frequency, modulation technique, and data encoding. To date, open source implementations of the PHY layer of a Z-Wave transceiver exist for R1 and R2, but not R3. The two major open source PHY layer implementations are from Scapy-Radio (2016) and KillerZee (2016). Scapy-Radio provides a working R2 configuration.

While KillerZee offers both R1 and R2 configurations, the frame reception rate is observed to be lower than the Scapy-Radio configuration. Upon analysis of the source code, the receive buffering mechanism may explain the low reception rate. When detecting an arriving frame, KillerZee is agnostic of the end of the frame and demodulates symbols until a fixed number of bytes are received. During the fixed-byte demodulation, a portion of another frame, such as an Acknowledgement (ACK), may also be received. While KillerZee looks for multiple frames in a receive buffer, it is unable to recover fragmented frames between fixed-byte demodulations. To account for the low ACK frame reception of KillerZee and lack of a working R1 configuration for Scapy-Radio, both implementations are used. Empirically, the majority of Z-Wave traffic is observed at R2, with R1 traffic being observed in special situations, including pairing operations.

The *SDR test platform* refers to a PC running both R1 and R2 SDR transceiver stacks. The stacks coexist on the same host computer, provided there is at least one USB3 and two other USB interfaces available. By utilizing disjoint User Datagram Protocol (UDP) ports, the test platform sends and receives Z-Wave frames using R1 and R2 configurations. Fig. 3 provides a portrait of SDR test platform used for the research herein.

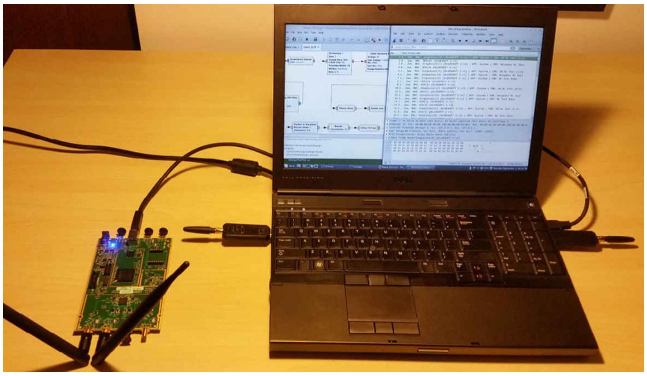


Fig. 3 – The SDR Test Platform: The Ettus B210 SDR is shown in the lower left. The laptop has two YARD Stick One SDR, shown on the left and right sides of the laptop.

3.2.1. R2 transceiver stack

An architecture diagram of the R2 transceiver stack is provided in Fig. 4. The Ettus B210 SDR, also known as a Universal Software Radio Peripheral (USRP), is used to provide a duplex interface between digitized I/Q baseband signals and analog RF signals. With regard to the receiver path, the B210 provides demodulated digital I/Q data of a filtered RF signal, received at the RX antenna, to a host computer using USB3. The USB3 data channel is regulated by the host using a USRP Hardware Driver (UHD) device driver 003.010.000. The I/Q data are passed to a GNUradio 3.7.5 layer, which is running the R2 transceiver provided by Scapy-Radio. The Scapy-Radio transceiver filters the digital baseband samples and demodulates the samples into symbols. It also provides preamble synchronization and Start of Frame (SOF) detection for Z-Wave Physical Protocol Data Units (PPDUs) at R2.

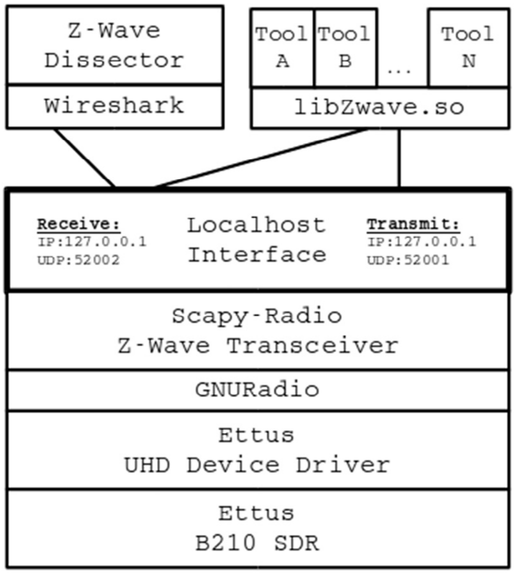


Fig. 4 – Protocol stack for SDR-based Z-Wave R2 transceiver.

The Z-Wave MPDUs are extracted from demodulated PPDUs by the Scapy-Radio transceiver and sent over the local interface of the host computer to UDP port 52002. Any application listening on this port may receive these Z-Wave frames. For this effort, the primary recipient includes Wireshark 2.0.1, with a custom dissector for Z-Wave frames encapsulated in the Scapy-Radio header. In addition, a custom shared library *libZwave.so* is developed so arbitrary applications may send, receive, and process frames as active participants of a Z-Wave network.

The transmit path is similar to the receive path, but in the opposite direction. Applications may transmit a Z-Wave MPDU by encapsulating it with a Scapy-Radio header and sending it over the local interface to UDP port 52001. The Scapy-Radio demodulator listens for arriving datagrams on this port and appends the PPDU header, consisting of a preamble and SOF, modulates the frame at baseband, and sends the digitized signal to the B210 SDR. The SDR upconverts the digital I/Q data to an analog RF signal and emits it through the TX antenna.

3.2.2. R1 transceiver stack

An architecture diagram of the R1 transceiver stack is provided in Fig. 5. The SDR for R1 traffic is the YARD Stick One. This SDR provides simplex communication, so two are required to provide duplex communications. The modulation and

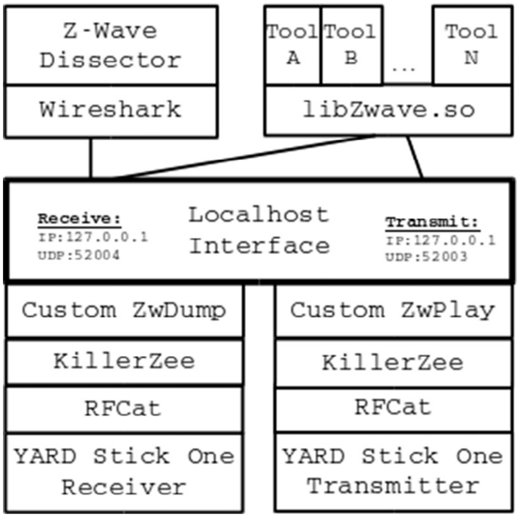


Fig. 5 – Protocol stack for SDR-based Z-Wave R1 transceiver.

demodulation occur on the SDR. RFCat provides data and control channels to the SDR. KillerZee 0.1 uses RFCat 1.0 to configure the YARD Stick One to send or receive Z-Wave frames at either R1 or R2. KillerZee also has an interface to the data channel of the radio using RFCat.

Two Python scripts are added to the R1 stack to make it compatible with the R2 stack. ZwDump is a tool provided in KillerZee, which is extended to take frames received over the radio, append a Scapy-Radio header, and send the frame over the local interface to UDP port 52004. ZwPlay is custom script added to KillerZee to listen on UDP port 52003. A frame received on this port is stripped of its Scapy-Radio header and sent to the SDR for transmission using the KillerZee interface. This scheme allows Wireshark and libZwave.so to send or receive R1 traffic without modification.

3.3. *Collection and injection techniques*

The SDR Test platform is used for passive collection and frame injection. For the demonstration topology, the SDR test platform is able to observe transmission of the entire network under study by exploiting the security vulnerabilities identified in Section 5. In addition to passive collection, the test platform is used to inject stimuli into the network under study. Tools are derived to allow byte values of Z-Wave frames to be manually constructed and injected into the network. Routed frames are manually constructed, injected into the network, and observed while being forwarded over the network under study. The observations lead to the identification of data field semantics, which are confirmed by additional frame injection activities.

In addition to simple frame injection, some aspects of the protocol are found to require more complex interaction. For example, to participate in frame forwarding, the test platform must receive, parse, update, and transmit the frame. The libZwave.so library implements a significant portion of the MAC and security layers. As routing details are discovered, these are also added to the library to facilitate more complex interactions between the SDR test platform and the network under study. Unless otherwise stated, the SDR test platform uses the node IDs of 10 and 11 in the source field of injected frames to be distinguishable from the legitimate nodes in the network.

3.4. *Reverse engineering process*

A direct approach to reverse engineering is to extract and statically analyze firmware, memory, and other non-volatile artifacts (Mesbah et al., 2017; Tellez et al., 2016). While it is possible to extract the firmware of Z-Wave devices, the reverse engineering challenges identified in Badenhop et al. (2016b), such as the proprietary special function registers, make it challenging to isolate the routing functionality using static analysis. A network protocol is inherently extrinsic, so the semantics may be derived by collecting and analyzing protocol messages (Matthies et al., 2015). In general, the forwarding and topology management aspects of the Z-Wave routing protocol are reverse engineered using a manual black-box analysis process and protocol fuzzing (Stutton et al., 2007), guided by available open-source literature such as Paetz (2013) and Badenhop et al. (2016b). The following steps are taken:

1. Normal protocol traffic is collected from the network under study.
2. Network traces are analyzed to identify differences in frames to isolate network protocol fields.
3. The semantics of the isolated fields are identified by injecting crafted frames into the network and observing the response. Through trial and error, the behavior of a field emerges. The semantics are confirmed by observing the network react as predicted to several variations of a given field.
4. Unknown fields are fuzzed, where frames are injected with arbitrary field values to discover semantics. Cases where a field's purpose remains hidden are otherwise noted herein.
5. Having identified a sufficient number of fields, the reverse engineering process focuses on the meaning of the messages containing the fields. Again, the SDR test platform injects targeted network messages into the network under study. The responses are observed and analyzed to estimate semantics. The semantics are confirmed by injecting similar frames into the network and observing predictable behavior.
6. Protocol transactions are realized as a composition of known network messages. The meaning of transactions are discovered and confirmed using the same techniques performed for fields and messages.

3.5. *Security analysis process*

Through reverse engineering, a model of the forwarding and topology management aspects of the Z-Wave routing protocol is realized. The security state of the model is evaluated *by inspection* (Andel and Yasinsac, 2007) for its susceptibility to classic integrity-based routing attacks performed by an outsider node. Device impersonation, manipulation of forwarded frames, and topology state corruption are all explored using the SDR test platform and network under study. Assuming the Z-Wave protocol is resistant to the identified integrity-based routing attacks, the goal of the security analysis activity is to discover violations to the assumptions. Similar to the reverse engineering process described in Section 3.4, violations are discovered interactively through an iterative process of injecting a given stimulus into the network, observing the network response, and analyzing the results to guide the subsequent experiments.

4. The reverse engineered Z-Wave routing protocol

The results of the reverse engineering analysis on the protocol are provided in the following sections. First, the forwarding mechanisms are presented. This is followed by details of topology and route management, including the data structures and protocol coordination messages.

4.1. *Source routing*

Routed Z-Wave frames use a network header, located between the MPDU header and application layer. Fig. 6 shows the fields of the network header. The first two bytes of the network header are divided into four 4-bit nibbles. The first nibble is the *failed hop*, used only for route error messages to declare the hop where the error occurred and is otherwise zero for other types of routed frames. The second nibble holds the SR type, which is used by routing nodes to determine how to forward the SR. The third nibble holds the length of SR in bytes. The fourth nibble holds the hop index field, which maintains the state of the SR while it is forwarded. The remaining bytes are the SR. The i^{th} byte in the SR is the node ID of the i^{th} hop in the route. The SR only contains the inner nodes of the route, relying on the MPDU header to provide the node IDs of the route endpoints. The SR is limited to four inner node hops (Paetz, 2013). Considering the implicit final hop to the destination, Z-Wave routes may be up to five hops in length.

4.1.1. *Forwarding behavior*

As with Dynamic Source Routing (DSR), the forwarding behavior of the inner nodes of a route is simple (Johnson et al., 2007). Since the routing layer is optional, the state of the *routed* flag in the control field of the MPDU header is used to resolve the presence of the network header (ITU G.9959, 2012). Upon receipt of a frame with this bit set, a node determines if it is responsible for forwarding the frame. The hop index field in the network frame provides the byte offset in the SR of the next hop. If a node's ID is located at this position, then the node updates the hop index field according to the route type field, recalculates the frame checksum, and retransmits the frame.

Table 1 – Forwarding targets for a SR of length N, $1 \le N \le 4$.

Hop index value	Next hop target
0 to $N-1$	In the SR, indexed by the hop index
N	Destination node in MPDU header
0xf	Source node in MPDU header

Table 1 summarizes which node is responsible for forwarding a routed frame as a function of the hop index, where *N* is the SR length. Since hop index is a byte offset, it is zero-based rather than a hop count. The destination realizes it is supposed to receive the frame when the hop index value matches the SR length field. While routed frames are normally forwarded to the destination, certain types of routing frames are reverse-routed to the source node. The source is able to recognize that it is the intended recipient when the hop index is 0xf. Since the hop index field is a 4-bit value, the field incurs an underflow when the node at hop index zero decrements the hop index before forwarding to the source.

4.1.2. *Routing frame types*

Three types of routing frames are summarized in Table 2. Application frames are routed from a source to destination using routing frame type 0x00. The other two routing frame types are used to inform the source node of the state of a routed application frame. A route ACK is issued by the destination node to confirm a given routed application frame is received. A route No-Acknowledgement (NACK) is issued by an inner node when a forwarding error is detected. The routing behavior for each type is described in the following subsections.

4.1.3. *Routed application frame*

When a source node needs to send a message to a target that is not a one-hop neighbor, it creates a routed application frame. The destination field of the MPDU header is set to the target of the frame and the routed bit is set. A network header is inserted between the MPDU header and application payload. The source node selects a route from a SR cache, embeds the SR

Table 2 – Known types of routing frames.

SR type value	Description	Hop index behavior
0x00	Application frame	Increments
0x03	Route ACK	Decrements
0x05	Route NACK	Decrements

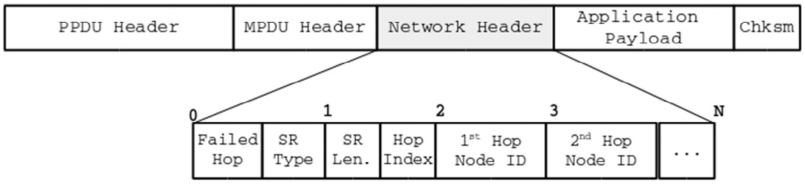
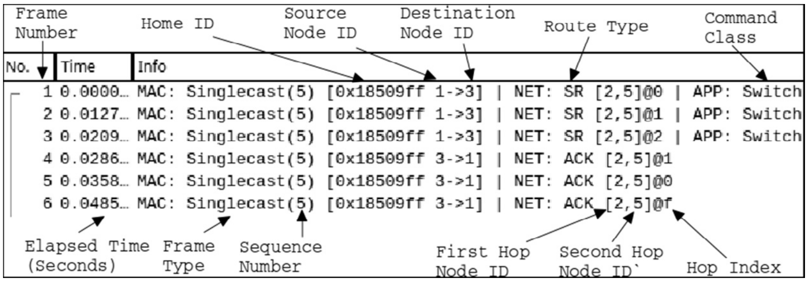


Fig. 6 – The Z-Wave Network Header format.



```
No.  Time     Info
 1 0.0000… MAC: Singlecast(5) [0x18509ff 1->3] | NET: SR [2,5]@0 | APP: Switch
 2 0.0127… MAC: Singlecast(5) [0x18509ff 1->3] | NET: SR [2,5]@1 | APP: Switch
 3 0.0209… MAC: Singlecast(5) [0x18509ff 1->3] | NET: SR [2,5]@2 | APP: Switch
 4 0.0286… MAC: Singlecast(5) [0x18509ff 3->1] | NET: ACK [2,5]@1
 5 0.0358… MAC: Singlecast(5) [0x18509ff 3->1] | NET: ACK [2,5]@0
 6 0.0485… MAC: Singlecast(5) [0x18509ff 3->1] | NET: ACK [2,5]@f
```

Fig. 7 – The frames observed in the network under study when a SR is routed from Node 1 to 3 and the corresponding route ACK taking the reverse route back to Node 1.

into the network header, and sets the route length field to the length of the selected route. The route type field is set to 0x00 to designate it as a routed application frame. The hop index field is set to zero to designate the next hop recipient as the first node identified in the SR. Finally, the source transmits the application frame. The first hop node identifies itself as responsible for forwarding the frame. Before retransmitting the frame, the hop index field is incremented to designate the node listed at the one byte offset of the SR as the next hop. The source and destination fields of the MPDU are not updated while the frame is forwarded. The destination realizes it is the recipient when it receives a routed application frame where the hop index is equal to the route length.

4.1.4. *Route acknowledgment*

A source node receives confirmation that a routed frame arrives at a destination node when it receives a route acknowledgment. Similar to MAC layer ACKs, the sender makes up to three attempts to send a routed application frame before giving up. Retries are conducted after failing to receive a route ACK within a certain time interval.

Upon receipt of a routed frame, the destination node prepares a route ACK. The route ACK MPDU header and network header are copied from the received application frame. The source and destination fields of the MPDU are swapped. The SR type field is set to 0x03 to designate it as a route ACK and the hop index is decremented before transmitting the route ACK. Without having to reverse the ordering of the SR in the header, the message traverses the reverse path of the SR by having each hop decrement the hop index field before retransmitting the frame. The source receives the route ACK when the hop index field is 0xf and uses the sequence number field in the MPDU header of the route ACK to identify the routed application frame being confirmed.

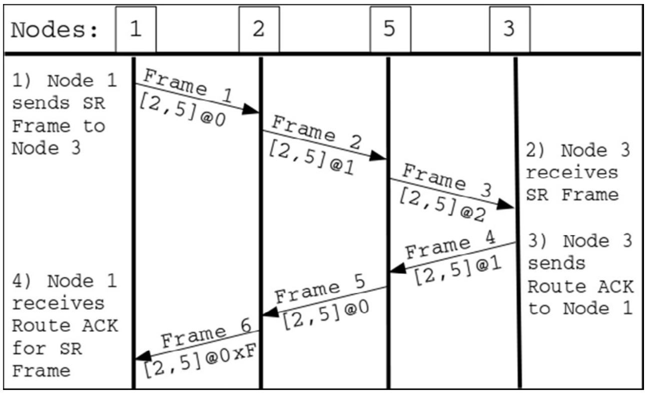
A routed application frame and associated route ACK are demonstrated on the network under study. The controller is invoked to toggle the switch state of Node 3 to ON. Fig. 7 shows the observed traffic. Frames 1–3 show the command message being routed from the controller to Node 3, where the hop index is incremented at each hop. Upon receipt of the command message, Node 3 sends a route ACK back to the controller. Frames 4–7 show the hop index decrementing as it is retransmitted, arriving back at the controller at frame 6.

To provide further clarity, Fig. 8 depicts the observed frames as a protocol activity timeline. The nodes along the SR are listed at the top. Each arrow corresponds to an observed frame in Fig. 7. The state of the source route is included below each frame. The figure shows the application frame being routed from Node 1 to 3. After receiving the routed application frame, Node 3 replies with a route ACK back to Node 1.

4.1.5. *Route error*

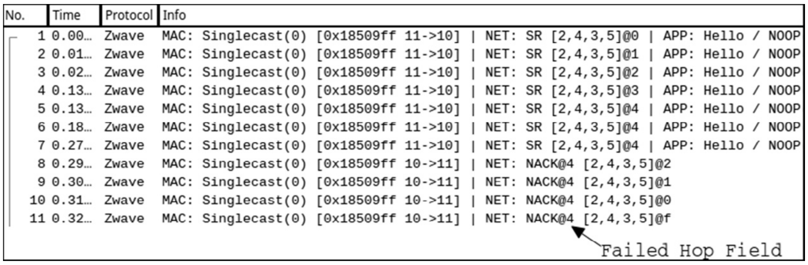
Nodes forwarding an application frame are responsible for detecting and reporting observed routing errors to the source node. After forwarding a routed frame, a node awaits confirmation that the frame is received by the next hop. In all cases except for the last hop, confirmation is made by witnessing the next hop appropriately forwarding the message to its next hop. Since the last inner hop node does not observe forwarding beyond the destination, it sets the *acknowledgment required* flag in the MPDU header before forwarding the frame to the destination. Receipt, in this case, is confirmed by receiving a MAC layer ACK from the destination. If no confirmation is observed in a given time interval, the node assumes the next hop failed to receive the frame and attempts to resend. After at least three attempts, the node gives up and reports a routing error.

The node discovering a routing error uses a route NACK message to notify the source node. The route NACK is similar to the route ACK because it is routed in reverse of the SR in the network header. The route NACK is composed of the MPDU and network headers of the offending frame, where the source and destination fields are swapped. The SR type field is set to 0x5 to designate it as a route NACK. The hop index of node



Nodes:
1
2
5
3
1) Node 1 sends SR Frame to Node 3
Frame 1 [2,5]@0
Frame 2 [2,5]@1
Frame 3 [2,5]@2
2) Node 3 receives SR Frame
3) Node 3 sends Route ACK to Node 1
Frame 4 [2,5]@1
Frame 5 [2,5]@0
Frame 6 [2,5]@0xF
4) Node 1 receives Route ACK for SR Frame

Fig. 8 – A protocol activity diagram of a SR routed from Node 1 to 3 and the route ACK taking the reverse route back to Node 1.



```
No. Time    Protocol Info
 1 0.00…  Zwave   MAC: Singlecast(0) [0x18509ff 11->10] | NET: SR [2,4,3,5]@0 | APP: Hello / NOOP
 2 0.01…  Zwave   MAC: Singlecast(0) [0x18509ff 11->10] | NET: SR [2,4,3,5]@1 | APP: Hello / NOOP
 3 0.02…  Zwave   MAC: Singlecast(0) [0x18509ff 11->10] | NET: SR [2,4,3,5]@2 | APP: Hello / NOOP
 4 0.13…  Zwave   MAC: Singlecast(0) [0x18509ff 11->10] | NET: SR [2,4,3,5]@3 | APP: Hello / NOOP
 5 0.13…  Zwave   MAC: Singlecast(0) [0x18509ff 11->10] | NET: SR [2,4,3,5]@4 | APP: Hello / NOOP
 6 0.18…  Zwave   MAC: Singlecast(0) [0x18509ff 11->10] | NET: SR [2,4,3,5]@4 | APP: Hello / NOOP
 7 0.27…  Zwave   MAC: Singlecast(0) [0x18509ff 11->10] | NET: SR [2,4,3,5]@4 | APP: Hello / NOOP
 8 0.29…  Zwave   MAC: Singlecast(0) [0x18509ff 10->11] | NET: NACK@4 [2,4,3,5]@2
 9 0.30…  Zwave   MAC: Singlecast(0) [0x18509ff 10->11] | NET: NACK@4 [2,4,3,5]@1
10 0.31…  Zwave   MAC: Singlecast(0) [0x18509ff 10->11] | NET: NACK@4 [2,4,3,5]@0
11 0.32…  Zwave   MAC: Singlecast(0) [0x18509ff 10->11] | NET: NACK@4 [2,4,3,5]@f
```

Fig. 9 – Captured frames as the result of the SDR platform injecting a SR frame into the network under study using imaginary source and destination IDs. Because Node 10 does not exist, Node 5 fails to forward the frame and generates a route NACK that takes the reverse route back to the source. The NACK frame indicates the fourth hop failed.

failing to confirm its receipt is copied to the failed hop field, the hop index field is decremented, and the frame is sent to begin traversal back to the source node. Upon receipt, the source node may use the failed hop field to select a new route from its SR cache that avoids the failed hop. As with a route ACK, the sequence number field in the MPDU header of the route NACK is used to associate the error with a recently transmitted frame.

To demonstrate a route NACK, the SDR test platform injects a single routed frame into the network under study. The source and destination fields are set to unused node IDs to force a routing error when the last hop of the route attempts to forward to the destination node. Fig. 9 shows the results of the frame capture during the route error event. In frames 1–5, the frame is routed over the inner nodes of the SR. Since the destination is fictitious, the last inner node never receives a MAC layer ACK of its forwarded frame. After three failed forwarding attempts, shown at frames 5–7, Node 5 generates a route NACK that is routed back to the source node in frames 8–11. The NACK in the figure identifies the fourth hop as the failed node, which is the destination node. A protocol activity timeline of the observed frames is provided in Fig. 10.

4.2. *Network management*

Fig. 11 depicts the data structures and activities necessary for topology maintenance and route discovery. Local topology information resides in each routing node as a one-hop NL. The global topology state resides in the adjacency table maintained by the controller. The adjacency table is updated by polling each node for its NL (Paetz, 2013). Each node also stores several caches of SRs. New routes are generated on demand by the controller, where a routing node makes a request for a SR to a given target. Having the global adjacency table, the controller is able to realize multiple routing solutions and provide them to the requesting node, where they are cached.

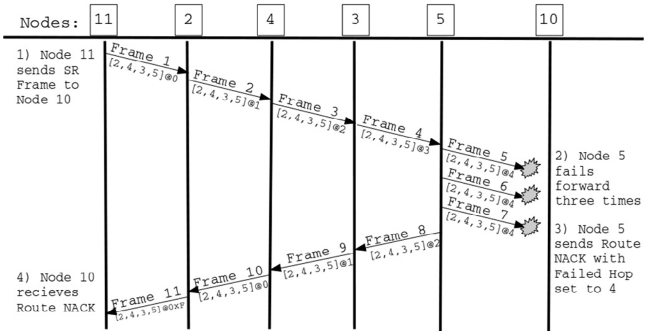


Fig. 10 – A protocol activity diagram of a SR routing from Node 11 to 10. Because Node 10 does not exist, Node 5 is unable to successfully forward the frame. It generates a route NACK, which takes the reverse route back to source Node 11.

In addition to a NL and several SR caches, routing nodes also have a backbone SR cache to be able to reach the controller. OpenZwave defines a route from the node to the controller as a *reverse route*; herein, it is designated as a *backbone route* to more appropriately describe its criticality. A node oblivious of backbone routes is unable to request new routes from the controller.

From a holistic perspective, the routing protocol has properties found in both reactive and proactive routing protocols. Like the proactive routing protocol Optimized Link Source Routing (OLSR), the Z-Wave routing protocol collects local topology state information from each node before any routes are needed. Like the reactive routing protocol DSR, routes are discovered at the request of the source node; however, the discovery is performed by the controller using its global topology state. Having aspects of both reactive and proactive routing protocols, Z-Wave can be considered a hybrid routing protocol. The following subsections describe the observed data structures, coordination messages, and transactions involved in maintaining topology state.

4.2.1. *Coordination messages*

Several coordination messages are identified as being associated with topology management and route discovery, which are summarized in Table 3. Note that all of the coordination messages belong to command class 0x01. This command class provides commands outside of the scope of network routing, including pairing operations, network and node ID assignments, and interactions between primary and secondary controllers. Open source literature such as OpenZwave (2016) do not name this command class, so it is designated herein as the *System* command class.

Several coordination messages utilize one of two identified *primitive* data structure formats. The first data structure is the NL primitive. It is used to convey topology state within topology management coordination messages, and its basic structure is shown in Fig. 12. The first byte of the primitive

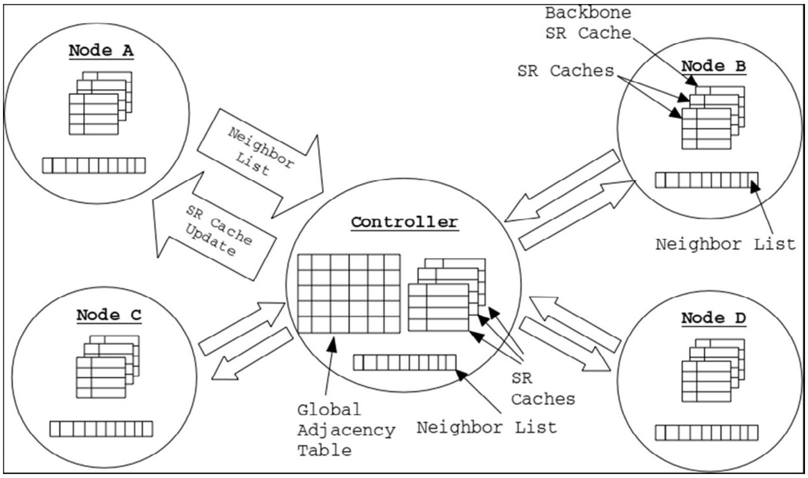


Fig. 11 – The Z-Wave Network Management Architecture.

indicates the length of the remaining bytes. The length field is followed by a variable length bitfield.

The structure of the bitfield is consistent with the record structure of the adjacency table (Badenhop et al., 2016b), where each bit corresponds to a particular node ID. The exact assignment is NodeID = $8i + j + 1$, $i = 0, \ldots, n - 1$; $j = 0, \ldots, 7$, where i is the byte offset from the first byte of the bitfield and n is the length of the bitfield in bytes. For each byte, j is the bit offset. The bits in each byte are big-endian and j is indexed from least to most significant bit. This is illustrated in Fig. 12 to show that the node IDs are not monotonically increasing when reading the bitfield from left to right.

The command byte of the message determines the implication of the bit value at each node ID position. For example, when a node provides its local NL to a controller, a high bit indicates that the sender is adjacent to the node ID corresponding with that bit position.

Table 3 – Known network commands of the System command class.

Command	Description	Parameters
0x04	Do NL test	NL primitive
0x05	Get NL	Target node ID
0x06	Report NL	NL primitive
0x07	NL test done	None
0x0C	SR cache assignment	SR cache entry primitive
0x14	Backbone cache assignment	SR cache entry primitive
0x15	SR request	Target node ID
0x18	NL test	Unknown parameter

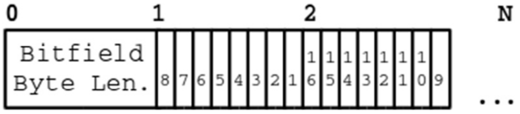


Fig. 12 – Neighbor list primitive containing the first 16 node IDs.

The second data structure is the SR cache entry primitive. It is used to exchange SR and its format is described in Fig. 13. The first byte designates the destination target of the SR. The second byte is composed of an upper and lower nibble. The upper nibble stores the cache entry index, which tells the recipient where to store the record in the cache region associated with the destination target. The lower nibble provides the length of the SR. This is followed by the variable length list of inner node hops of the SR.

The last byte of the primitive has an impact on route selection; however, only two values are observed. When this byte is 0x10, the route is used by the recipient node. When the byte is 0x08, some devices avoid using the route entirely. Other devices give an entry with a status of 0x08 a lower priority, using it only after routes marked with a status of 0x10 are exhausted. While the behavior is consistent with a status or priority indicating field, the true meaning of this field may not be discovered until the firmware is fully reverse engineered.

4.2.2. *NL updates*

A node updates its NL based on the results of NL tests, initiated upon the receipt of a `Do NL Test` message. The payload of this message contains a NL primitive, which describes how the tests are to be conducted. A set bit in the NL primitive instructs the node to perform a NL test on the corresponding node ID. When the node encounters a bit that is not set, this tells the node to remove that node ID from its NL. Thus, messages with a NL primitive of a string of zero bits that covers all node IDs in the network clears the node's NL. Alternatively, a string of set bits covering all node IDs of the network forces the recipient to perform a NL test on every node ID.

A node conducts a NL test on a given target using a `Do NL Test` message. The node sends this message to the target using

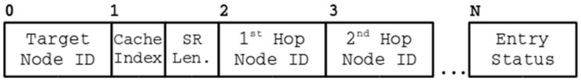
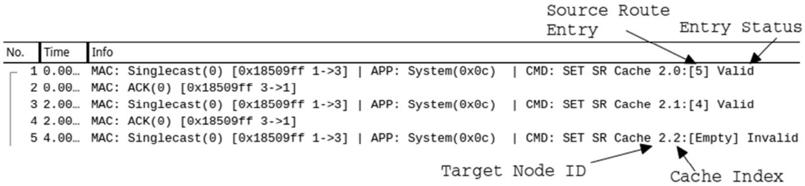


Fig. 13 – SR cache entry primitive.



```
No. Time    Info
1 0.00…  MAC: Singlecast(0) [0x18509ff 1->3] | APP: System(0x0c)  | CMD: SET SR Cache 2.0:[5] Valid
2 0.00…  MAC: ACK(0) [0x18509ff 3->1]
3 2.00…  MAC: Singlecast(0) [0x18509ff 1->3] | APP: System(0x0c)  | CMD: SET SR Cache 2.1:[4] Valid
4 2.00…  MAC: ACK(0) [0x18509ff 3->1]
5 4.00…  MAC: Singlecast(0) [0x18509ff 1->3] | APP: System(0x0c)  | CMD: SET SR Cache 2.2:[Empty] Invalid
```

Fig. 14 – SR cache assignments for Node 3.

rate configuration R1, regardless of the *de facto* PHY rate configuration. The NL test passes if the target replies with an ACK. The test fails if, after three attempts, no ACK is received. The test message has a single byte as a parameter; however, its meaning is not yet known. Once a node has performed a NL test on every requested node, it sends a `NL Test Done` to the sender of the `Do NL Test` message.

4.2.3. *Adjacency table updates*

The adjacency table is updated when a controller queries a node for its NL. The query is performed by sending a `Get NL` message to a target. The target replies with a `Report NL` message, which contains the node's NL. The format of the NL primitive is conveniently consistent with the adjacency table (Badenhop et al., 2016b). Updating the adjacency table is as simple as performing a direct memory copy of the NL primitive to the memory location holding the adjacency record of the sender.

The controller adjacency table is updated when the device is rebooted or in response to a user invoked request to repair the network. Unsolicited `Report NL` messages are ignored by the controller of the network under study to prevent external manipulation of the adjacency table structure.

4.2.4. *Route request*

A node may request a route to a target using the `SR Request` message, where the destination node ID of the route is the only parameter. The response to this request depends on the capabilities of the recipient. A controller replies with a sequence of `SR Cache Assignment` messages in *cache entry index* order. The routes also appear to be in shortest-length-first order. The controller may also follow these messages with a set of `Backbone Cache Assignment` messages to freshen the node's cache of critical routes to the controller.

If a `SR Request` is received by a routing node that is not a controller, the node replies to the sender with its backbone cache as a sequence of `Backbone Cache Assignment` messages. Having an updated backbone cache, the requesting node may now query routes from the controller. The behavior provides a recovery mechanism for nodes with stale or corrupted backbone SR caches.

The `SR Request` message is discovered through protocol fuzzing of command byte values for the *System* command class. While the SDR test platform can cause a node to facilitate a route request, the activity is not observed during normal operations of the network under study. As a result, the conditions necessary for a legitimate node to invoke a route request are not yet known.

A node is capable of caching multiple routes for each target. By injecting `SR Request` messages into the network under study, observations of the resulting cache assignment messages have never exceeded four entries per target. Without dynamic and static analysis of the device firmware, further aspects of the cache remain hidden.

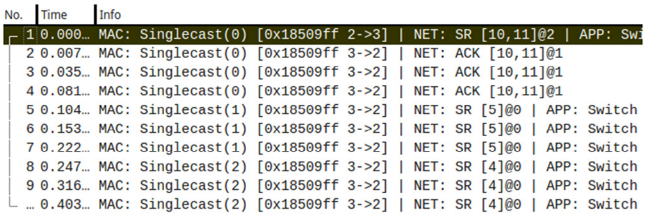
4.2.5. *Route selection*

A node consults its SR cache when it requires a multihop route. When more than one solution is available, the source node must make a selection. If the selected route fails, the source node must select an alternative route. The selection process is repeated until the cache solutions are exhausted or the source node gives up. The route selection behavior is studied by isolating a node from the network under study and causing it to attempt to route an application frame to a target. As the source node incurs routing failures, it selects alternative routes and eventually concedes. The observations are compared with the node's SR cache for the given target to reveal the route selection behavior.

For the experiment, Node 3 is the isolated node that attempts to send a routed application frame to Node 2. Fig. 14 shows three cache assignment messages sent to Node 3 prior to the injected frame. The first cache entry holds a path to Node 2 through Node 5. The second cache entry holds a route to Node 2 through Node 4. The third entry is empty and marked invalid.

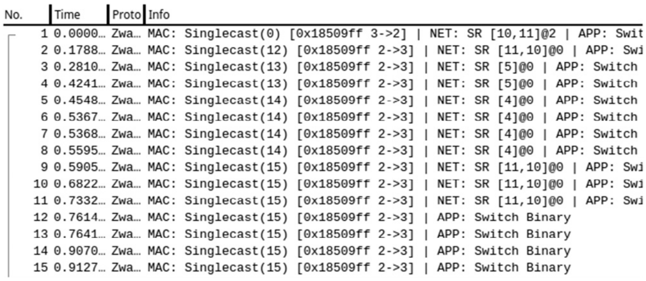
Fig. 15 shows the reaction of Node 3 in response to the injected frame and after the other nodes are disabled. The first frame in the capture is the frame sent by the SDR test platform, which spoofs as Node 2 asking Node 3 for its state. After replying with a route ACK, Node 3 attempts to send the application layer response through Node 5 in frames 5–7, which is the first cache entry for Node 2. After not receiving a route ACK from Node 2, it makes three attempts to route through Node 4, which is the second cache entry for that target. After this fails, it gives up.

The experiment is repeated; however, Node 2 becomes the isolated node attempting to send an application frame to Node 3 over the network. The SR cache in Node 2 for Node 3 is identical to the SR cache in Node 3 for Node 2, as shown in Fig. 14. With all nodes disabled except for Node 2, the SDR test platform



```
No. Time     Info
1 0.000…  MAC: Singlecast(0) [0x18509ff 2->3] | NET: SR [10,11]@2 | APP: Swi
2 0.007…  MAC: Singlecast(0) [0x18509ff 3->2] | NET: ACK [10,11]@1
3 0.035…  MAC: Singlecast(0) [0x18509ff 3->2] | NET: ACK [10,11]@1
4 0.081…  MAC: Singlecast(0) [0x18509ff 3->2] | NET: ACK [10,11]@1
5 0.104…  MAC: Singlecast(1) [0x18509ff 3->2] | NET: SR [5]@0 | APP: Switch
6 0.153…  MAC: Singlecast(1) [0x18509ff 3->2] | NET: SR [5]@0 | APP: Switch
7 0.222…  MAC: Singlecast(1) [0x18509ff 3->2] | NET: SR [5]@0 | APP: Switch
8 0.247…  MAC: Singlecast(2) [0x18509ff 3->2] | NET: SR [4]@0 | APP: Switch
9 0.316…  MAC: Singlecast(2) [0x18509ff 3->2] | NET: SR [4]@0 | APP: Switch
… 0.403…  MAC: Singlecast(2) [0x18509ff 3->2] | NET: SR [4]@0 | APP: Switch
```

Fig. 15 – Route selections for Node 3 (Aeon Appliance Switch).



```
No.  Time     Proto Info
  1 0.0000… Zwa… MAC: Singlecast(0) [0x18509ff 3->2] | NET: SR [10,11]@2 | APP: Swit
  2 0.1788… Zwa… MAC: Singlecast(12) [0x18509ff 2->3] | NET: SR [11,10]@0 | APP: Swi
  3 0.2810… Zwa… MAC: Singlecast(13) [0x18509ff 2->3] | NET: SR [5]@0 | APP: Switch
  4 0.4241… Zwa… MAC: Singlecast(13) [0x18509ff 2->3] | NET: SR [5]@0 | APP: Switch
  5 0.4548… Zwa… MAC: Singlecast(14) [0x18509ff 2->3] | NET: SR [4]@0 | APP: Switch
  6 0.5367… Zwa… MAC: Singlecast(14) [0x18509ff 2->3] | NET: SR [4]@0 | APP: Switch
  7 0.5368… Zwa… MAC: Singlecast(14) [0x18509ff 2->3] | NET: SR [4]@0 | APP: Switch
  8 0.5595… Zwa… MAC: Singlecast(14) [0x18509ff 2->3] | NET: SR [4]@0 | APP: Switch
  9 0.5905… Zwa… MAC: Singlecast(15) [0x18509ff 2->3] | NET: SR [11,10]@0 | APP: Swi
 10 0.6822… Zwa… MAC: Singlecast(15) [0x18509ff 2->3] | NET: SR [11,10]@0 | APP: Swi
 11 0.7332… Zwa… MAC: Singlecast(15) [0x18509ff 2->3] | NET: SR [11,10]@0 | APP: Swi
 12 0.7614… Zwa… MAC: Singlecast(15) [0x18509ff 2->3] | APP: Switch Binary
 13 0.7641… Zwa… MAC: Singlecast(15) [0x18509ff 2->3] | APP: Switch Binary
 14 0.9070… Zwa… MAC: Singlecast(15) [0x18509ff 2->3] | APP: Switch Binary
 15 0.9127… Zwa… MAC: Singlecast(15) [0x18509ff 2->3] | APP: Switch Binary
```

Fig. 16 – Route selections for Node 2 (GE Outdoor Switch).

injects the same frame and the observed behavior is shown in Fig. 16. In frame 2, Node 2 initially selects the reverse route of the received frame as the SR of the reply message. After the reverse route fails once, it uses its first and second cache entries. After they fail, it makes several more attempts to use the reverse route of the request in frames 9–11. The last three frames of the figure show Node 2 attempting to reach Node 3 locally, where a SR is not used. Although both nodes have the same route entries in their SR cache, the route selection behavior for Node 2 is different from Node 3. The difference may be explained because Node 2, the GE switch, has a newer library and protocol version than Node 3, as reported in Section 3.1.

5. Security analysis on Z-Wave routing protocol

In this section, the integrity and reliability implications of Z-Wave the routing protocol are analyzed. Several vulnerabilities are discovered on the network under study using the SDR test platform.

5.1. *Existing security in Z-Wave*

Z-Wave utilizes several security mechanisms, which include cryptographic, policy-driven, behavior detection, and out of band mechanisms. They are summarized in the following subsections.

5.1.1. *Security command class*

Z-Wave provides a confidentiality, source integrity, and data integrity service through the *Security* command class. Application frames may be encapsulated in a security frame that is both encrypted and signed. The frame is secured through symmetric encryption using AES and three shared keys, known by every node of the network requiring the security service. The operations of the *Security* command class are described in detail in Fouladi and Ghanoun (2013) and Badenhop and Ramsey (2016).

During network inclusion, the device specifies which of its supported command classes must use the security layer. When an application layer command is exchanged for one of the designated command classes, it must be embedded within a secure frame. Otherwise, the frame is ignored by the device (Badenhop and Ramsey, 2016).

Empirical evidence shows that the utilization of the security service is proportional to the purpose of the device. For example, physical security related devices such as door locks, motion detectors, and alarms are more likely to use the security service than smart energy devices such as light bulbs, appliance switches, and thermostats.

The *Security* command class has particular limitations when used. First, only the application layer message is encrypted. The fields in the MPDU header, network layer, and security frame header are not encrypted. Data integrity is provided through signature checking, which only applies to the fields being signed. In addition to the application payload, this includes the MPDU source node ID, destination node ID, and MPDU length field (Badenhop and Ramsey, 2016). Note that none of the network header fields are involved in the signature, so these fields may be modified in transit without causing the signature check to fail. Since all paired devices use the same authentication key, source integrity at the node ID resolution is not possible. Instead, devices may only discriminate between messages originating from trusted and untrusted sources.

While none of the devices in the network under study utilize the *Security* command class, the Yale Z-Wave door lock analyzed in Badenhop and Ramsey (2016) possesses this capability. While the door lock silently drops door unlock commands sent by the SDR test platform, it responds to *System* command class messages. The implication is that the *System* command class is not required to use the security layer.

5.1.2. *Out of band triggers*

Many of the critical activities lack a way to be initiated over the network. Instead, they require physical proximity and human intervention for invocation. For example, pairing operations require the user to reset a device, press a button, or reinsert batteries to force a node into pairing mode. Another example of an out of band protection involves updates to the adjacency table in the controller. Since unsolicited NL reports are ignored, an attacker attempting to corrupt the topology state must react to a request from the controller. Known ways of causing a controller to update its adjacency table include rebooting the device or requesting a network healing action through the controller's user interface. Ways of forcing these events through the Z-Wave network remain unknown; however, network healing may be susceptible to the rogue controller vulnerability found in (Fuller and Ramsey, 2015). If Z-Wave devices are powered through Z-Wave switches, the switches may be remotely toggled to induce a reset event. While the Z-Wave serial API has the ability to reset a Z-Wave transceiver (OpenZwave Defs.h, 2016; Badenhop et al., 2016b), an attack vector to utilize the API through the Z-Wave network has not yet been discovered.

5.1.3. *Privileged controller*

The controller is privileged because it possesses both global topology information and node capability information for each node that is paired to the network. A controller may identify and react to integrity-based attacks by correlating events with its global knowledge. Conversely, non-controller nodes are less secure because they do not have access to global information necessary to identify inconsistencies. For example, the controller ignores messages sent from the SDR test platform if the source ID is of a node that is never paired to the network. The other nodes in the network under study are unable to recognize outsider nodes. As a result, they respond to unsolicited frames from arbitrary sources so long as the source node ID is in a valid range (i.e., 1 to 232) (ITU G.9959, 2012).

5.1.4. *Watchdog routing*

Route error messages are generated by a forwarding node when it fails to observe the next hop forwarding the message within a time interval. The node observing the forwarding node is known as a *watchdog*. Having each previous hop monitor the protocol compliance of the next hop assists in the identification of malicious or non-cooperative forwarding nodes within the network (Bhalaji and Shanmugam, 2009; Hao et al., 2004; Tiwari et al., 2009). The capabilities of watchdog routing for Z-Wave are examined in Section 5.2.5.

5.2. *Vulnerabilities*

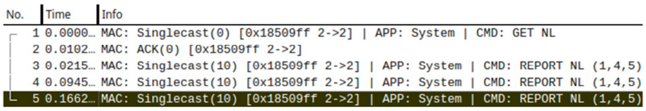
The Z-Wave protocol, as observed in the network under study, is analyzed for security weaknesses in data and source integrity. The results are described in the following subsections.

5.2.1. *Impersonation*

Impersonation attacks violate the source integrity of the protocol. With the exception of the controller, Z-Wave devices implicitly trust the source and destination fields of the MPDU frame. This makes it trivial to impersonate frames originating from the controller or another device. To demonstrate impersonation, a single frame is injected into the network using the SDR test platform. The injected frame is a request; observing a response to the forged request implies that the replying node accepts the message as legitimate.

The injected frame is a `Get NL` message to Node 2. The source ID field in the MPDU header is also set to Node 2 to make it clear that the message originates from an artificial source. The assumption is that a node will not have to use its radio to query and respond to itself for data residing in its memory. The frame is emitted into the network and the responding frames are observed in Fig. 17.

The injected frame is shown as frame 1. While Node 2 acknowledges the forged frame, this only means that the MPDU header and checksum checks are valid. Proof of the impersonation vulnerability is in the remaining frames, where Node 2 attempts to tell itself its NL in response to the unsolicited request. An interesting observation is that Node 2 acknowledges frame 1 but does not acknowledge frames 3, 4, and 5. Although each of these frames has Node 2 as the destination, the node is able to avoid acknowledging frames it actually sends. This capability may be extended to identify Sybil attacks occurring on the network, where a node alerts the controller when it receives a message from itself that it did not send.



```
No. Time      Info
  1 0.0000... MAC: Singlecast(0) [0x18509ff 2->2] | APP: System | CMD: GET NL
  2 0.0102... MAC: ACK(0) [0x18509ff 2->2]
  3 0.0215... MAC: Singlecast(10) [0x18509ff 2->2] | APP: System | CMD: REPORT NL (1,4,5)
  4 0.0945... MAC: Singlecast(10) [0x18509ff 2->2] | APP: System | CMD: REPORT NL (1,4,5)
  5 0.1662... MAC: Singlecast(10) [0x18509ff 2->2] | APP: System | CMD: REPORT NL (1,4,5)
```

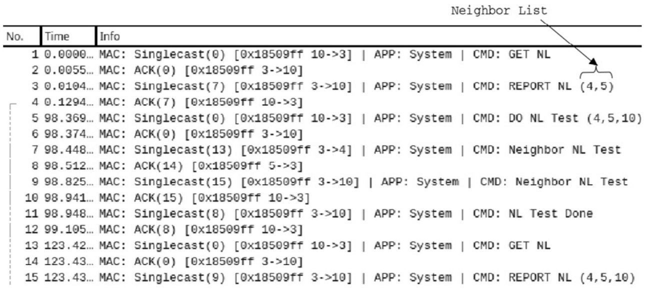
Fig. 17 – SDR impersonates Node 2 by requesting the NL of Node 2, to which Node 2 replies.

Devices using the Z-Wave security layer have some protection against outsider impersonation. Devices specify, at a command class granularity, which command messages must use the secure frame. Secure frames are signed and encrypted using keys exchanged during network inclusion. An outsider who is not in possession of the authentication and encryption keys is unable to transmit a valid secure frame. Regardless of the chosen source ID, the outsider is unable to impersonate the origin of a command message if the destination requires that it is sent in a secure frame. However, the outsider may still perform impersonation attacks on the device using commands from a supported command class that is not required by the device to use the security layer.

5.2.2. *Arbitrary NL modification*

Routing nodes, including the controller, are vulnerable to having their NLs manipulated by an external source. This is shown using the SDR test platform in the network under study. Acting as outsider Node 10, the SDR test platform interacts with Node 3 to add itself to the NL of Node 3. This is performed in two steps. First, a process is executed on the SDR test platform to listen and acknowledge received frames at R1 sent to Node 10. Second, a separate process on the laptop sends Node 3 a `Do NL Test` message with Node 10 set as the target. This initiates the NL add activity, which is captured using Wireshark. Fig. 18 shows the combined results of the activity at R1 and R2.

In frame 1, the NL is requested from Node 3 to demonstrate that Node 10 is currently not a neighbor. The NL is reported at frame 3, having Nodes 4 and 5 as one-hop neighbors. The `Do NL Test` is sent by the SDR test platform at frame 5, where the targets include the original NL and Node 10. Frames 7–10 are received at R1 and involve Node 3 testing each requested node from lowest to highest node ID. Frame 7 shows Node 3 testing if Node 4 is a one-hop neighbor; however, due to the poor reception rate of the R1 SDR stack, the response is not captured. The test frame sent to Node 5 is not captured either; however, frame 8 shows Node 5 responding to the test request message. Fortunately, a complete test is captured in frames 9 and 10. In frame 9, Node 3 is testing Node 10 as a one-hop neighbor. The SDR test platform responds with an ACK at R1 in frame 10. Node 3 announces that the tests have completed in frame 11. The NL of Node 3 is again requested by Node 10 in frame 13. The response shows that its NL now includes Node 10, along with the original neighbors of Node 4 and 5.



```
No. Time      Info
  1 0.0000... MAC: Singlecast(0) [0x18509ff 10->3] | APP: System | CMD: GET NL
  2 0.0055... MAC: ACK(0) [0x18509ff 3->10]
  3 0.0104... MAC: Singlecast(7) [0x18509ff 3->10] | APP: System | CMD: REPORT NL (4,5)
  4 0.1294... MAC: ACK(7) [0x18509ff 10->3]
  5 98.369... MAC: Singlecast(0) [0x18509ff 10->3] | APP: System | CMD: DO NL Test (4,5,10)
  6 98.374... MAC: ACK(0) [0x18509ff 3->10]
  7 98.448... MAC: Singlecast(13) [0x18509ff 3->4] | APP: System | CMD: Neighbor NL Test
  8 98.512... MAC: ACK(14) [0x18509ff 5->3]
  9 98.825... MAC: Singlecast(15) [0x18509ff 3->10] | APP: System | CMD: Neighbor NL Test
 10 98.941... MAC: ACK(15) [0x18509ff 10->3]
 11 98.948... MAC: Singlecast(8) [0x18509ff 3->10] | APP: System | CMD: NL Test Done
 12 99.105... MAC: ACK(8) [0x18509ff 10->3]
 13 123.42... MAC: Singlecast(0) [0x18509ff 10->3] | APP: System | CMD: GET NL
 14 123.43... MAC: ACK(0) [0x18509ff 3->10]
 15 123.43... MAC: Singlecast(9) [0x18509ff 3->10] | APP: System | CMD: REPORT NL (4,5,10)
```

Fig. 18 – SDR is adding fake Node 10 to the NL of Node 3.

Thus, a fake node is added as a neighbor to the NL of a routing node.

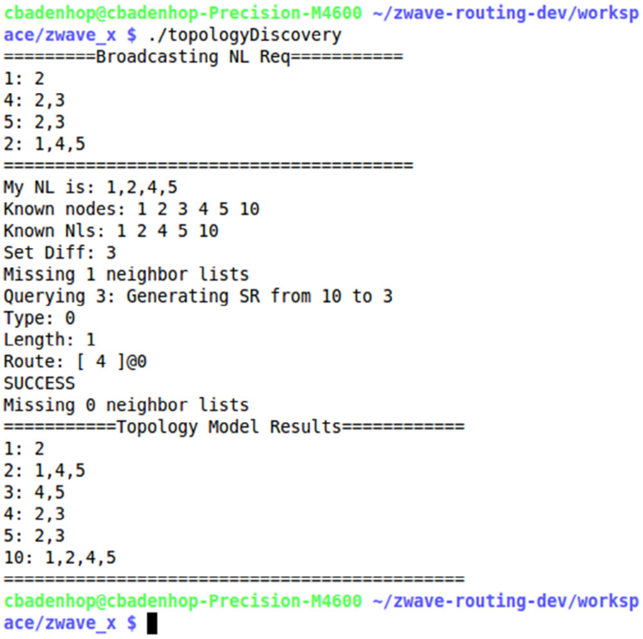
This attack may be used to add nodes into the topology without formally conducting network inclusion with the controller. The possibility that a modified NL ends up in the adjacency table depends on the behavior of the controller. The controller used in this network under study sends a `Do NL Test` message before sending a `Get NL` request and only sets the IDs of the nodes that have performed network inclusion in the NL primitive of the NL test message. Thus, rogue insertions get masked out of the target's NL by the controller during initialization and network healing operations.

The attack requires the outsider node to be one hop from the target node. If the outsider is more than one hop away, it is unable to successfully ACK the NL test frame sent over R1. Attempts to route MAC layer ACK frames are silently dropped by the forwarding nodes.

5.2.3. *Outsider topology discovery*

While the outsider may passively observe routing frames over a long period of time to realize the topology of the target network, a more direct approach is to use the NL request message to learn the NL of every node. Unlike the controller, an outsider may not be aware of every node in the network. Nodes more than one hop from the outsider require a SR to reach them; however, constructing a SR requires an existing topology model. These problems are addressed using the Topology Discovery algorithm shown in Algorithm 1. In this algorithm, the outsider learns of all of its one-hop neighbors by first broadcasting a `NL Request` message. The outsider collects NL reports of its one-hop neighbors to construct a two-hop topology. In turn, each node in the two-hop topology that has not provided its NL is queried to learn of its neighbors. The outsider uses the two-hop topology model to realize valid SRs to send `NL Request` messages to each target. The responses are added to the two-hop topology model to create a three-hop topology model. Again, the new nodes discovered in the three-hop topology model are queried for their NLs to expand the three-hop model to a four-hop topology model. The algorithm continues until a NL for every discovered node is acquired. Upon termination and assuming the network is not partitioned, the derived topology model represents the target Z-Wave network. Moreover, the outsider realizes its location within the network topology.

Algorithm 1 is implemented in C++ using *libZwave.so* to interact with the network under study. SRs are derived from the topology model using *Dijkstra's Shortest Path* algorithm. Fig. 19 provides the console output of the algorithm running on the SDR test platform as Node 10, physically located within the network. The figure is divided into three regions. The top region shows the results of the outsider deriving its NL using a local broadcast request, of which Nodes 1, 4, 5, and 2 provide replies. In the middle section, the algorithm realizes that Node 3 is the only observed node in the two-hop topology model without a known NL. The algorithm sends a request message using a SR derived from the two-hop topology model that routes through Node 4. The NL report from Node 3 indicates it is adjacent to Nodes 4 and 5. After the topology model is updated, the algorithm terminates because every discovered node has provided its NL.



```
cbadenhop@cbadenhop-Precision-M4600 ~/zwave-routing-dev/worksp
ace/zwave_x $ ./topologyDiscovery
=========Broadcasting NL Req===========
1: 2
4: 2,3
5: 2,3
2: 1,4,5
=======================================
My NL is: 1,2,4,5
Known nodes: 1 2 3 4 5 10
Known Nls: 1 2 4 5 10
Set Diff: 3
Missing 1 neighbor lists
Querying 3: Generating SR from 10 to 3
Type: 0
Length: 1
Route: [ 4 ]@0
SUCCESS
Missing 0 neighbor lists
===========Topology Model Results============
1: 2
2: 1,4,5
3: 4,5
4: 2,3
5: 2,3
10: 1,2,4,5
============================================
cbadenhop@cbadenhop-Precision-M4600 ~/zwave-routing-dev/worksp
ace/zwave_x $ █
```

Fig. 19 – Results of the SDR test platform performing the Topology Discovery algorithm on the network under study.

Algorithm 1 The **Outsider Topology Discovery** algorithm, where *OutsiderNL*, *KnownNodes*, *HaveNLs*, *MissingNLs* are each a set of node IDs. The topology state is stored in *top*, which is a collection of NLs and is indexed by the owner of the NL. The variable *msg* is a Z-Wave `NL report` frame, where *msg.src* is the source ID and *msg.nl* is the neighbor list included in the frame. The variable *OutsiderID* is a node ID the outsider designates as itself.

```
$KnownNodes \leftarrow \{OutsiderID\}$
$HaveNLs \leftarrow \{\emptyset\}$
$OutsiderNL \leftarrow \{\emptyset\}$
Broadcast_NL_Request()
while !timeout() do
    $msg \leftarrow Receive\_NL\_Report()$
    $OutsiderNL \leftarrow \{msg.src\} \cup OutsiderNL$
    $top[msg.src] \leftarrow msg.nl$
    $HaveNLs \leftarrow \{msg.src\} \cup HaveNLs$
    $KnownNodes \leftarrow \{msg.src\} \cup KnownNodes$
    for all $i \in msg.nl$ do
        $KnownNodes \leftarrow \{i\} \cup KnownNodes$
    end for
end while
$top[OutsiderID] \leftarrow OutsiderNL$
$HaveNLs \leftarrow \{OutsiderID\} \cup HaveNLs$
$NeedNLs \leftarrow KnownNodes - HaveNLs$
while $NeedNLs \neq \{\emptyset\}$ do
    for all $i \in NeedNLs$ do
        Send_NL_Request(top,i)
        $msg \leftarrow Receive\_NL\_Report(i)$
        assert($msg.src = i$)
        $top[msg.src] \leftarrow msg.nl$
        $HaveNLs \leftarrow \{msg.src\} \cup HaveNLs$
        for all $j \in msg.nl$ do
            $KnownNodes \leftarrow \{j\} \cup KnownNodes$
        end for
    end for
    $NeedNLs \leftarrow KnownNodes - HaveNLs$
end while
return top
```

The bottom portion of the figure reports the derived topology model as a list of adjacencies in node ID order. The learned adjacencies between legitimate nodes are consistent with Fig. 2, which defines the demonstration topology of the network. Since the reported topology model includes the adjacencies for Node 10, the outsider is able to launch multihop attacks over the network.

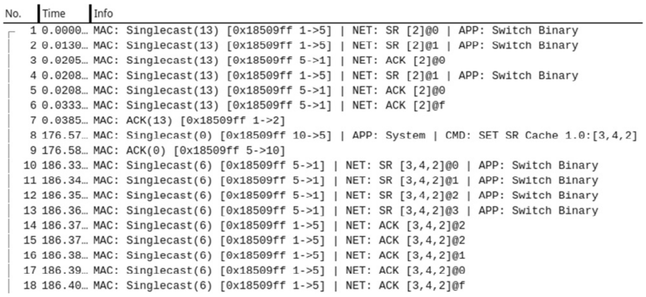
5.2.4. *Arbitrary SR cache modification*

Due to the centralized command and control structure of the routing protocol, nodes are assigned routes using `SR Cache Assignment` messages. The outsider may use this message to change the routing behavior of target source nodes. The vulnerability is demonstrated on the network under study, and the results are shown in Fig. 20. The figure initially shows an application frame taking a route through Node 2 in frames 1–7. In frame 8, the first entry of the source node's routing cache for Node 5 is modified by the outsider. In frames 10–19, another application frame is sent from Node 1 to Node 5. The figure shows the application frame taking the route provided by the outsider.

5.2.5. *Modification of routed frames*

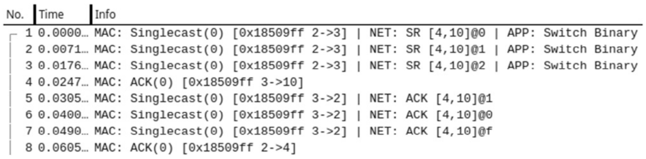
Using the SR cache modification attack, the outsider can direct application frames to flow through a Man-In-The-Middle (MITM) node, where they may be modified. The extent of the manipulation is contingent on the satisfaction of the watchdog node observing the MITM node. Failing to adequately forward a frame results in the watchdog alerting the frame originator with a route NACK. Upon receipt of the NACK, the source node may select an alternative route that does not include the MITM node. Thus, the outsider must avoid invoking any route NACKs to remain stealthy and maximize the lifetime of its advantage (Badenhop and Mullins, 2014). For this purpose, several manipulations are explored to identify conditions where the frame is forwarded by a MITM node without tripping the watchdog.

The case without frame modification is shown in Fig. 21. In the figure, a forged routed frame is injected into the network from Node 2 to Node 3, using a route that traverses through Nodes 4 and 10. Acting as the outsider node, the SDR test platform correctly forwards the frame to the destination without alerting the watchdog (i.e., Node 4). The watchdog is satisfied because Node 10 only modifies the hop count field and recalculates the checksum before forwarding. Note that Node 10 is also able to correctly decrement and forward the route ACK in frames 5 and 6 without alerting the watchdog.



```
No. Time     Info
 1 0.0000… MAC: Singlecast(13) [0x18509ff 1->5] | NET: SR [2]@0 | APP: Switch Binary
 2 0.0130… MAC: Singlecast(13) [0x18509ff 1->5] | NET: SR [2]@1 | APP: Switch Binary
 3 0.0205… MAC: Singlecast(13) [0x18509ff 5->1] | NET: ACK [2]@0
 4 0.0208… MAC: Singlecast(13) [0x18509ff 1->5] | NET: SR [2]@1 | APP: Switch Binary
 5 0.0208… MAC: Singlecast(13) [0x18509ff 5->1] | NET: ACK [2]@0
 6 0.0333… MAC: Singlecast(13) [0x18509ff 5->1] | NET: ACK [2]@f
 7 0.0385… MAC: ACK(13) [0x18509ff 1->2]
 8 176.57… MAC: Singlecast(0) [0x18509ff 10->5] | APP: System | CMD: SET SR Cache 1.0:[3,4,2]
 9 176.58… MAC: ACK(0) [0x18509ff 5->10]
10 186.33… MAC: Singlecast(6) [0x18509ff 5->1] | NET: SR [3,4,2]@0 | APP: Switch Binary
11 186.34… MAC: Singlecast(6) [0x18509ff 5->1] | NET: SR [3,4,2]@1 | APP: Switch Binary
12 186.35… MAC: Singlecast(6) [0x18509ff 5->1] | NET: SR [3,4,2]@2 | APP: Switch Binary
13 186.36… MAC: Singlecast(6) [0x18509ff 5->1] | NET: SR [3,4,2]@3 | APP: Switch Binary
14 186.37… MAC: Singlecast(6) [0x18509ff 1->5] | NET: ACK [3,4,2]@2
15 186.37… MAC: Singlecast(6) [0x18509ff 1->5] | NET: ACK [3,4,2]@2
16 186.38… MAC: Singlecast(6) [0x18509ff 1->5] | NET: ACK [3,4,2]@1
17 186.39… MAC: Singlecast(6) [0x18509ff 1->5] | NET: ACK [3,4,2]@0
18 186.40… MAC: Singlecast(6) [0x18509ff 1->5] | NET: ACK [3,4,2]@f
```

Fig. 20 – The routing behavior of Node 5 is modified by the SDR test platform to take an alternate route.

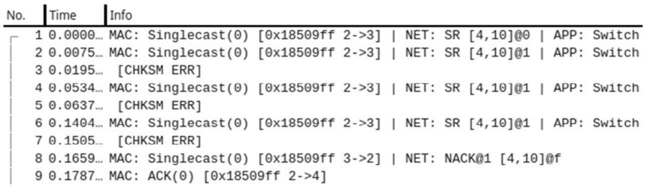


```
No. Time     Info
 1 0.0000… MAC: Singlecast(0) [0x18509ff 2->3] | NET: SR [4,10]@0 | APP: Switch Binary
 2 0.0071… MAC: Singlecast(0) [0x18509ff 2->3] | NET: SR [4,10]@1 | APP: Switch Binary
 3 0.0176… MAC: Singlecast(0) [0x18509ff 2->3] | NET: SR [4,10]@2 | APP: Switch Binary
 4 0.0247… MAC: ACK(0) [0x18509ff 3->10]
 5 0.0305… MAC: Singlecast(0) [0x18509ff 3->2] | NET: ACK [4,10]@1
 6 0.0400… MAC: Singlecast(0) [0x18509ff 3->2] | NET: ACK [4,10]@0
 7 0.0490… MAC: Singlecast(0) [0x18509ff 3->2] | NET: ACK [4,10]@f
 8 0.0605… MAC: ACK(0) [0x18509ff 2->4]
```

Fig. 21 – The SDR test platform is forwarding routed frames without error.

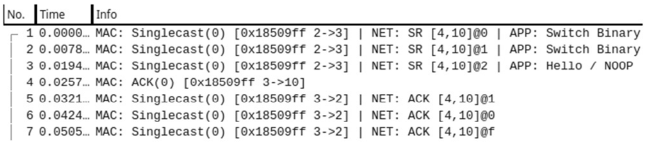
A counter example is provided in Fig. 22. Upon receiving frame 2, Node 10 correctly updates the hop index field but fails to recalculate the checksum before forwarding. While the destination receives frame 3, it does not provide an ACK because the received frame does not pass the checksum test. As the watchdog, Node 4 also detects the checksum error and resends the forwarded frame to Node 10 two additional times. Node 10 makes additional transmission attempts in frames 5 and 7. Since the checksum is invalid, these are also ignored by the recipient. After observing three failures, Node 4 generates a route NACK in frame 8, reporting Node 10 (i.e., hop index 1) as the offender.

An outsider node can modify the application payload without triggering the watchdog. With respect to the application layer of a frame, the first two bytes are the most significant, defining the command class and the command to be executed, respectively. By modifying these bytes, the intent of the frame is drastically changed. To show this on the network under study, the MITM node is modified to write nulls to the first two bytes, turning any frame into a harmless `Hello` message. Again, the test frame is injected into the network and the response is captured in Fig. 23. Frame 3 shows that Node 10 forwards the injected frame but changes its meaning to be a `Hello` message. Node 3 replies with a route ACK, allowing Node 2 to believe Node 3 received the switch state update. Meanwhile, the watchdog does not detect the tampering of the application layer, so a route NACK is not generated.



```
No. Time     Info
 1 0.0000… MAC: Singlecast(0) [0x18509ff 2->3] | NET: SR [4,10]@0 | APP: Switch
 2 0.0075… MAC: Singlecast(0) [0x18509ff 2->3] | NET: SR [4,10]@1 | APP: Switch
 3 0.0195…  [CHKSM ERR]
 4 0.0534… MAC: Singlecast(0) [0x18509ff 2->3] | NET: SR [4,10]@1 | APP: Switch
 5 0.0637…  [CHKSM ERR]
 6 0.1404… MAC: Singlecast(0) [0x18509ff 2->3] | NET: SR [4,10]@1 | APP: Switch
 7 0.1505…  [CHKSM ERR]
 8 0.1659… MAC: Singlecast(0) [0x18509ff 3->2] | NET: NACK@1 [4,10]@f
 9 0.1787… MAC: ACK(0) [0x18509ff 2->4]
```

Fig. 22 – A route NACK is triggered when the SDR test platform forwards a frame with an invalid checksum.



```
No. Time     Info
 1 0.0000… MAC: Singlecast(0) [0x18509ff 2->3] | NET: SR [4,10]@0 | APP: Switch Binary
 2 0.0078… MAC: Singlecast(0) [0x18509ff 2->3] | NET: SR [4,10]@1 | APP: Switch Binary
 3 0.0194… MAC: Singlecast(0) [0x18509ff 2->3] | NET: SR [4,10]@2 | APP: Hello / NOOP
 4 0.0257… MAC: ACK(0) [0x18509ff 3->10]
 5 0.0321… MAC: Singlecast(0) [0x18509ff 3->2] | NET: ACK [4,10]@1
 6 0.0424… MAC: Singlecast(0) [0x18509ff 3->2] | NET: ACK [4,10]@0
 7 0.0505… MAC: Singlecast(0) [0x18509ff 3->2] | NET: ACK [4,10]@f
```

Fig. 23 – MITM Attack: Changing the meaning of a forwarded frame without detection.

Several other fields are modified by the outsider node to test the detection capability of the watchdog, with mixed results. It is discovered that the MITM node may modify the destination or next inner node hop without triggering the watchdog. Conversely, modifying the hop count to have a value beyond the SR length is detected and reported by the watchdog. A rigorous study on the MPDU header, network layer, and application layer fields is required to completely understand the detection rules of the watchdog.

In addition to the watchdog, the Z-Wave security layer, described in Section 5.1.1, allows a destination to determine if several fields of a secure frame have been tampered with while being forwarded. Consequently, a MITM modifying the payload or destination field of a secured application frame is detected by the destination upon checking the signature of the frame before the application message is processed. Given the destination finds the frame to be tampered, it does not generate a route ACK. After a timeout is observed by the source node, it retransmits the secure frame. If the MITM attack continues to modify the secure frame, consecutive timeouts allow the source node to conclude the route is flawed. Eventually, the source node selects an alternative SR from its cache. Depending on the topology, the new route may not include the MITM node.

5.2.6. *Black Hole attack*

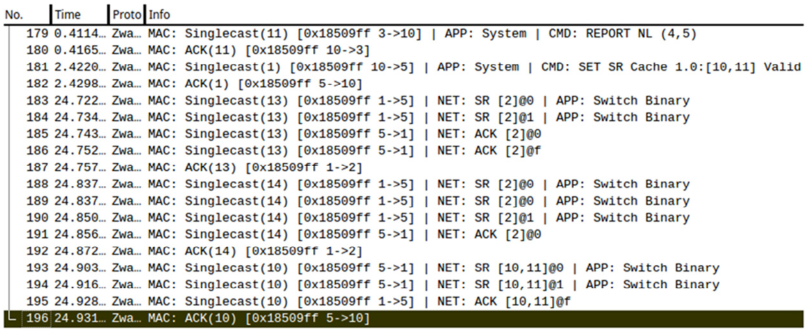
By exploiting the impersonation, topology discovery, watchdog subversion, and SR cache modification vulnerabilities, a Black Hole attack may be conducted on a Z-Wave network for a given source and destination pair. A Black Hole attack is a frame dropping attack where a node under the influence of the attacker, a Black Hole Node (BHN), silently drops application frames when it is expected to forward them (Badenhop et al., 2016a). Active Black Hole attacks exacerbate frame loss by manipulating topology state information in the network to increase the number of routes that flow through the BHN (Badenhop and Mullins, 2014). For protocols where routing options are competitive, such as Ad-hoc On-Demand Distance Vector (AODV) and DSR, the BHN may exaggerate its connectedness to other nodes to provide a more lucrative option than those realized through legitimate route discovery.

The Black Hole attack predicate in Badenhop et al. (2016b) states that a Black Hole attack is possible if the shortest hop distance path between the source node and BHN is less than the shortest hop distance path between the source and destination node. With respect to Z-Wave, routes are not competitive; rather, they are assigned by the controller. In this case, the predicate is relaxed to where a Black Hole attack is possible if there exists at least one path between the BHN and the source that does not contain the destination node. The path may not contain the destination node because, after appending the destination node to the end of the route, the resulting route is not loop-free. Due to the route length constraint of Z-Wave routing, this path must be no longer than four hops; otherwise, the BHN is unable to append the destination to the end of the route.

To avoid the watchdog of the BHN reporting frames being dropped, the BHN may insert a fictitious hop between the BHN and the destination in the cache assignment message it sends to the source node. In this way, the watchdog is able to observe the BHN forwarding the routed frame as expected without improper modification. At this point, the BHN is supposed to serve as the watchdog for the protocol. Instead of alerting the source to the failure, the BHN allows the frame to be silently lost by the fictitious node. To conduct this variation of the Black Hole attack, a route must exist between the BHN and source node that is no more than three hops and does not contain the destination node. This allows both the BHN and imaginary node to be appended as inner hops of the assigned SR to the destination.

The BHN initiates the attack on the network under study by performing topology discovery to derive a local adjacency table. The table is used to determine if a path exists between the source node and the BHN that is no more than three hops in length and does not contain the destination node. If a path meeting these criteria is found, a `SR Cache Assignment` message is sent to the source node, where the inner hops in the cache entry primitive include all inner hops from the source node to the BHN, the BHN itself, and imaginary Node 11. The target field of the cache assignment message is set to the destination node ID, the cache index is set to zero, and the status byte is set to 0x10 to give the route the highest preference when the source node selects a route to the destination.

In Fig. 24, the attack is demonstrated for a route from Node 5 to Node 1 on the network under study. Prior to the first frame, the BHN (i.e., Node 10) performs the Topology Discovery algorithm. In frame 181, the BHN updates the SR cache of Node 5 using a route that contains both Nodes 10 and 11. Using the human–controller interface, the switch of Node 5 is remotely toggled to invoke network traffic between Nodes 1 and 5. The



```
No.  Time      Proto Info
 179 0.4114… Zwa… MAC: Singlecast(11) [0x18509ff 3->10] | APP: System | CMD: REPORT NL (4,5)
 180 0.4165… Zwa… MAC: ACK(11) [0x18509ff 10->3]
 181 2.4220… Zwa… MAC: Singlecast(1) [0x18509ff 10->5] | APP: System | CMD: SET SR Cache 1.0:[10,11] Valid
 182 2.4298… Zwa… MAC: ACK(1) [0x18509ff 5->10]
 183 24.722… Zwa… MAC: Singlecast(13) [0x18509ff 1->5] | NET: SR [2]@0 | APP: Switch Binary
 184 24.734… Zwa… MAC: Singlecast(13) [0x18509ff 1->5] | NET: SR [2]@1 | APP: Switch Binary
 185 24.743… Zwa… MAC: Singlecast(13) [0x18509ff 5->1] | NET: ACK [2]@0
 186 24.752… Zwa… MAC: Singlecast(13) [0x18509ff 5->1] | NET: ACK [2]@f
 187 24.757… Zwa… MAC: ACK(13) [0x18509ff 1->2]
 188 24.837… Zwa… MAC: Singlecast(14) [0x18509ff 1->5] | NET: SR [2]@0 | APP: Switch Binary
 189 24.837… Zwa… MAC: Singlecast(14) [0x18509ff 1->5] | NET: SR [2]@0 | APP: Switch Binary
 190 24.850… Zwa… MAC: Singlecast(14) [0x18509ff 1->5] | NET: SR [2]@1 | APP: Switch Binary
 191 24.856… Zwa… MAC: Singlecast(14) [0x18509ff 5->1] | NET: ACK [2]@0
 192 24.872… Zwa… MAC: ACK(14) [0x18509ff 1->2]
 193 24.903… Zwa… MAC: Singlecast(10) [0x18509ff 5->1] | NET: SR [10,11]@0 | APP: Switch Binary
 194 24.916… Zwa… MAC: Singlecast(10) [0x18509ff 5->1] | NET: SR [10,11]@1 | APP: Switch Binary
 195 24.928… Zwa… MAC: Singlecast(10) [0x18509ff 1->5] | NET: ACK [10,11]@f
└ 196 24.931… Zwa… MAC: ACK(10) [0x18509ff 5->10]
```

Fig. 24 – Black Hole attack on Z-Wave network where Node 10 is dropping frames sent from Node 5 to Node 1.

toggle switch command is routed to Node 5 in frames 183–187. To be sure that the command is honored, Node 1 requests the switch state of Node 5 in frames 188–192. Node 5 generates a reply and routes it through the BHN in frames 193–196. The Black Hole attack is observed at frame 194, where the BHN forwards the frame to non-existent Node 11. To close the loop with the source node, the BHN forges the route ACK from the destination in frame 195 of the figure. Being satisfied that Node 1 received its update, Node 5 makes no attempts to resend its state information using an alternative route.

The scope of the attack is limited to the network layer. While frames are silently dropped, the application layer expects a response to its query and may repeatedly attempt to resend the request. Not shown in Fig. 24 is that the controller makes several repeated attempts to query the state of Node 5 every 40 seconds. Since no routing errors have been detected, the source node continues to use the route containing the BHN. While the BHN successfully drops the reply to each repeated request, the human controller interface may eventually notify the user of the communication problems with the remote node. This, in turn, may provoke the use of network healing in attempt to remedy the failure.

5.3. *Recommendations*

Common to all of the discovered vulnerabilities is the exploitation of the hierarchical relationship between the controller and routing nodes. The issue is that routing nodes rely on external direction from a controller or arbitrary neighbor that is trusted without authentication. As a result, the topology and routes can be arbitrarily modified to benefit the attacker. For example, the Black Hole attack may be used on these networks to prevent home security sensors from reporting intrusion events or drop frames that would otherwise actuate an alarm to alert the user. Given these vulnerabilities, several improvements may be made to the protocol.

The primary recommendation is to mandate the use of the *Security* command class for all *System* command class messages. This prevents an outsider who does not have the AES keys from forging messages to control the topology. At a minimum, this policy increases the cost of routing attacks, requiring physical access to a device EEPROM to extract the encryption and authentication keys (Badenhop and Ramsey, 2016).

Given the available global topology information, the controller should be the only node to send SR cache assignments to other devices. Unfortunately, devices are unable to authenticate at the node ID resolution with the current symmetric key capability. A second recommendation is to move to an asymmetric key system, where each node has a public and private authentication key. The intent for Z-Wave is to be a low cost home automation system (Paetz, 2013), so a full Public Key Infrastructure (PKI) may not be feasible. At a minimum, the controller should use an asymmetric key pair so that devices may authenticate messages from the controller. This allows the policy to be enforced, where a device updates its cache only if the source authenticated assignment message originates from the controller. As with the network key (Badenhop and Ramsey, 2016), the controller's public key may be exchanged with a device during the pairing operation.

In the event that a full PKI implementation exists for Z-Wave, more secure routing is possible. For example, following the practices found in Secure Ad-hoc On-Demand Vector (SAODV), the destination may sign the route ACK to make forgery more difficult (Guerrero-Zapata, 2006). Moreover, each hop of a source route may, in turn, sign the message to ensure the SR, selected by the source node, is taken. Hop by hop authentication is already found in the Ariadne routing protocol (Andel and Yasinsac, 2007; Hu et al., 2002), which is a more secure form of DSR.

6. Conclusion

Being a reverse engineering effort, a complete understanding of the routing protocol, even if achievable, is resource intensive. There are several aspects of the routing protocol still requiring reverse engineering and security analysis. Future work should examine the role of secondary controllers, the existence of a route discovery mechanism similar to DSR as reported in Paetz (2013), multipath routing, Z-Wave to IP gateway activities, and *System* command class messages not identified in Table 3. Once the proprietary I/O mechanisms of the Z-Wave transceiver chip are identified, static analysis of the firmware may reveal further details of the routing protocol.

As of 2014, new Z-Wave devices may support *Z-Wave Plus*, which is an extended capability beyond the standard Z-Wave protocol. The Z-Wave Alliance has revealed little about the implications of the capability other than it is to provide "a whole new level of smart home capabilities" using a new hardware platform (Z-Wave Plus, 2016). To date, the availability of open source literature and analysis tools, such as an R3 SDR transceiver, is limited for the new technology. None of the devices in the network under study support Z-Wave Plus, so it is not yet known if the vulnerabilities discovered herein extend to the new hardware platform. Until Z-Wave Plus devices become ubiquitous and analysis tools become available, the evaluation of the security implications of Z-Wave Plus is left as future work.

The work performed herein is a result of an observational study on a limited set of Z-Wave devices. While the contributions of this work provide supporting evidence that the Z-Wave protocol has security flaws, conclusive evidence requires a randomized experiment where devices are randomly sampled from the market. Simply evaluating more devices results in a larger observational study with a marginal improvement in scope; therefore, a randomized experiment should be conducted as future work to determine if the discovered vulnerabilities are limited to a particular vendor, firmware version, topology configuration, or hardware version.

The Black Hole attack provided in this work applies to a single source and destination pair. Future work should extend this capability to provide a network-wide attack, which attempts to maximize the number of source and destination pairs affected by the Black Hole attack. With a network-wide attack capability, the applicability of the analytical Black Hole attack model to predict frame loss in Z-Wave networks, developed in Badenhop and Mullins (2014) and Badenhop et al. (2016b), may be explored.

In this paper, several contributions have been made regarding the security implications of the Z-Wave routing protocol for IoT applications. The forwarding and topology mechanisms of the Z-Wave routing protocol are reverse engineered using passive and active observations on a real-world Z-Wave network. A security assessment of this protocol on the network under study reveals that the hierarchical relationship between the routing nodes and controller may be exploited by a malicious outsider. Not only can an outsider impersonate other nodes, it also has the ability to discover and manipulate the topology, modify route caches of nodes, and manipulate frames in transit. The vulnerabilities are exploited to conduct a Black Hole attack on the network under study, where it is shown that frames are silently discarded for a given source and destination. The Black Hole attack can be used to prevent sensor reports or actuating commands between the controller and devices, inhibiting the functionality of the IoT automation system. The results of the security analysis suggest the Z-Wave routing protocol is vulnerable to integrity-based attacks; however, a more rigorous study is required to characterize the classes of Z-Wave devices possessing the identified vulnerabilities.

Acknowledgments

The views expressed in this article are those of the authors and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government. This article is approved for public release under case number 88ABW-2016-5241.

REFERENCES

Andel T, Yasinsac A. Surveying security analysis techniques in MANET routing protocols. IEEE Commun Surv Tut 2007;9(4):70–84.

Badenhop C, Mullins B. A black hole attack model using topology approximation for reactive ad-hoc routing protocols. Int J Secur Netw 2014;9(2):63–77.

Badenhop C, Ramsey B. Carols of the Z-Wave security layer; or, robbing keys from Peter to unlock Paul. Int J PoC 2016;12:6–12.

Badenhop C, Fuller J, Hall J, Ramsey B, Rice M. Evaluating ITU-T G.9959: wireless systems in the critical infrastructure. In: Butts J, Shenoi S, editors. Critical infrastructure protection IX, IFIPS WG 11.10, vol. 9. Springer; 2015. p. 61–79.

Badenhop C, Ramsey B, Mullins B. An analytical black hole attack model using a stochastic topology approximation technique for reactive ad-hoc routing protocols. Int J Netw Secur 2016a;18(4):667–77.

Badenhop C, Ramsey B, Mullins B, Malloux L. Extraction and analysis of non-volatile memory of the ZW0301 module, a Z-Wave transceiver. Digit Invest 2016b;17:14–27.

Bhalaji N, Shanmugam A. Association between nodes to combat blackhole attack in DSR based MANET. In: WOCN 2009 proceedings of the sixth international conference on wireless and optical communications networks. 2009. p. 403–7.

Bihl T, Bauer K, Temple M, Ramsey B. Dimensional reduction analysis for physical layer device fingerprints with application to Zigbee and Z-Wave devices. In: MILCOM. 2015. p. 360–5.

Fouladi B, Ghanoun S. 2013. Security evaluation of the Z-Wave wireless protocol. Presented at Blackhat USA.

Fuller J, Ramsey B. Rogue Z-Wave controllers: a persistent attack channel. In: SenseApp. IEEE; 2015. p. 734–41.

Fuller J, Ramsey B, Rice M, Pecarina J. Misuse-based detection of Z-Wave network attacks. Comput Secur 2017;64:44–58.

Guerrero-Zapata M. 2006. Secure On-Demand Distance Vector (SAODV) Routing.

Hall J, Ramsey B. 2016. Breaking bulbs briskly by bogus broadcasts. Presented at ShmooCon, Washington, DC.

Hall J, Ramsey B, Rice M, Lacey T. Z-Wave network reconnaissance and transceiver fingerprinting using software-defined radios. In: International conference on cyber warfare and security. 2016. p. 163–71.

Hao Y, Haiyun L, Fan Y, Songwu L, Lixia Z. 2004. Security challenges in mobile ad hoc networks: challenges and solutions. IEEE Wireless Communications.

Hu YC, Perrig A, Johnson DB. Ariadne: a secure on-demand routing protocol for ad hoc networks. In: The 8th annual international conference on mobile computing and networking. 2002. p. 12–23.

ITU G.9959, 2012. ITU Recommendation G.9959: short range narrow-band digital radiocommunication transceivers – PHY and MAC layer specifications.

Johnson D, Hu Y, Maltz D. 2007. The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks.

KillerZee. Github – joswr1ght/killerzee: Killerzee: tools for attacking and evaluating z-wave networks; 2016. Available from: https://github.com/joswr1ght/killerzee. [Accessed 21 April 2017].

Matthies C, Pirl L, Azodi A, Meinel C. Beat your mom at solitaire – a review of reverse engineering techniques and countermeasures. In: 2015 6th IEEE international conference on software engineering and service science (ICSESS). 2015. p. 1094–7.

Mesbah A, Lanet JL, Mezghiche M. Reverse engineering a java card memory management algorithm. Comput Secur 2017;66:97–114.

Narayanaswamy S, Kawadia V, Sreenivas RS, Kumar PR. Power control in ad-hoc networks: theory, architecture, algorithm and implementation of the compow protocol. In: European wireless conference. 2002. p. 156–62.

OpenZwave. Open Z-Wave homepage; 2016. Available from: http://www.openzwave.com/home. [Accessed 21 April 2017].

OpenZwave Defs.h. Openzwave defs.h; 2016. Available from: http://www.openzwave.com/dev/Defs_8h_source.html. [Accessed 21 April 2017].

Paetz C. Z-Wave basics. Lexington, KY: CreateSpace Independent Publishing Platform; 2013.

Patel J, Ramsey B. Comparison of parametric and non-parametric statistical features for Z-Wave fingerprinting. In: MILCOM. 2015. p. 378–82.

Scapy-Radio. Github – bastilleresearch/scapy-radio: scapy-radio (from original hg repo); 2016. Available from: https://github.com/BastilleResearch/scapy-radio. [Accessed 21 April 2017].

Stutton M, Greene A, Amini P. Fuzzing: brute force vulnerability discovery. Upper Saddle River, NJ: Addison-Wesley; 2007.

Tellez M, El-Tawab S, Heydari MH. IoT security attacks using reverse engineering methods on WSN applications. In: 2016 IEEE 3rd world forum on Internet of Things (WF-IoT). 2016. p. 182–7.

Tiwari M, Arya V, Choudhari R, Choudhary K. Designing intrusion detection to detect black hole and selective forwarding attack in WSN based on local information. In: International conference on computer sciences and convergence information technology. 2009. p. 824–8.

Vera. Vera: smarter home control; 2016. Available from: http://getvera.com. [Accessed 21 April 2017].

Z-Stick. Z-Wave USB stick; 2016. Available from: http://aeotec.com/z-wave-usb-stick. [Accessed 21 April 2017].

Z-Wave Plus. Z-Wave plus; 2016. Available from: http://www.zwaveproducts.com/learn/z-wave/z-wave-plus. [Accessed 21 April 2017].

Zwave Protocol 1. Z-Wave protocol stack – Z-Wave protocol layer basics; 2016. Available from: http://www.rfwireless-world.com/Tutorials/z-wave-protocol-stack.html. [Accessed 21 April 2017].

Zwave Protocol 2. Understanding Z-Wave networks, nodes & devices; 2016. Available from: http://www.vesternet.com/resources/technology-indepth/understanding-z-wave-networks. [Accessed 21 April 2017].

Christopher W. Badenhop is a PhD student of computer engineering at the Air Force Institute of Technology. He received a Masters in cyberspace operations from the Air Force Institute of Technology in 2012 and a Masters in computer engineering from Wright State University in 2006. His research interests include computer network security, embedded system security, reverse engineering, and RF communication.

Scott R. Graham is an assistant professor of computer engineering at the Air Force Institute of Technology. He received the PhD degree in electrical engineering from the University of Illinois at Urbana-Champaign in 2004. His research interests center on cyber physical systems, looking at the interaction of computer architecture, networks, and security for critical infrastructure protection.

Benjamin W. Ramsey is a former faculty member at the Air Force Institute of Technology. He received the PhD degree in computer science from the Air Force institute of Technology in 2014. His research interests include wireless network security and critical infrastructure protection.

Barry E. Mullins is a professor of computer engineering at the Air Force Institute of Technology. He received the PhD degree in electrical engineering from Virginia Polytechnic Institute and State University in 1997. His research interests include cyber operations, critical infrastructure protection, computer/network/embedded systems security, wired/wireless networking, and reverse code engineering.

Logan O. Mailloux, CISSP, CSEP (BS 2002, MS 2008, PhD 2015) is a commissioned officer in the United States Air Force and assistant professor at the Air Force Institute of Technology. His research interests include system security engineering, complex information systems, and quantum based cyber security systems. He is a member of Tau Beta Pi, Eta Kappa Nu, INCOSE, ITEA, the ACM, and IEEE.