We Have A Deal: we provide the lego bricks, you build cool wireless attacks

Damien Cauquil[1] and Romain Cayre[23]
dcauquil@quarkslab.com
rcayre@laas.fr

[1] Quarkslab
[2] CNRS, LAAS, 7 avenue du colonel Roche, F-31400
[3] Univ de Toulouse, INSA, LAAS, F-31400

Abstract. Attacks on wireless protocols are as numerous as the tools used to perform them, each being tied to a protocol and implementing one or more attacks. This fragmentation hinders interoperability and code reuse, impacting security research and leading to more fragmentation as researchers need to reinvent the wheel or adapt existing code to other hardware.

In this paper, we standardize attacks on wireless protocols as a combination of eleven attack primitives. We detail how these attack primitives helped us shape a corresponding tool set and how we leverage them to perform complex attacks on real-world wireless networks. We present the design of these tools and the mechanism used to combine them as well as the pros and cons of the chosen mechanisms compared to the theoretical approach.

Finally, we demonstrate how these attack primitives simplify the security analysis of wireless protocols, like *Meshtastic*, and enable complex attack workflows, illustrated on *RF4CE* and *Logitech Unifying*, enabling researchers to develop simple tools that can be combined with our tool set.

1 Introduction

Wireless protocols have become critical for most connected systems as they are used to create communication networks, interconnect multiple systems, transmit information from one device to another, with a lot of different usages. These protocols rely on wireless signals to send and receive data, and on cryptography to ensure confidentiality, authenticity and integrity when supported. They are also prone to attacks due to some weaknesses introduced during their design or errors during the development of specific implementations, allowing remote attackers to take complete control of a device, leak sensitive information or even disrupt communications.

Attacks on wireless protocols are still evolving, security researchers trying to figure out new ways of exploiting more and more complex vulnerabilities, helped by quickly evolving and newly created wireless protocols. In fact, wireless protocols have improved a lot these last years: some initiatives like *Thread* define new protocols built upon other protocols while others simply continue to evolve at a fast pace like Bluetooth Low Energy (BLE).

1.1 Wireless attacks and tools fragmentation

Multiple wireless attacks have been designed and implemented on various protocols, leading to many specific security tools like the *aircrack-ng* suite [1], *gnuradio* [14], *Ubertooth* [11], *InternalBlue* [17], *Killerbee* [29], *MouseJack* [20], *Keykiriki* [36], *Gattacker* [33], *crackle* [31], *Btlejuice* [6], *Btlejack* [5], *Mirage* [7] or *Injectable* [8] to name a few. These tools are for most of them used with specific hardware required to interact with the radio signals related to the target protocol. In some cases this hardware is available off-the-shelf like a Wi-Fi network adapter or a Bluetooth adapter, while in other cases it may be very specific and require a firmware to be installed in it.

This fragmentation causes a lot of issues, including but not limited to:

— each tool uses its own hardware,
— attacks implemented in a tool are not portable to another tool or hardware,
— researchers keep reinventing the wheel (host/interface communication protocol, packet processing, etc.).

1.2 Reducing fragmentation

As a first response to this fragmentation issue, we started working in August 2021 on *WHAD* [30], an attempt at providing researchers and users with a flexible framework able to interact with various wireless protocols.

During the development of *WHAD*, we spent quite some time thinking about the different tools we may include in this framework. Some of them were more than obvious while others were quite difficult to define, and it became clear that we needed to think these tools as *Lego* bricks that we could combine to create complex behaviors. This is how the idea of defining a systematization of wireless attacks came into our minds and slowly made its way, and even if it first looked like a difficult task, we quickly realized we have already started this work during the development.

Fundamentally, wireless attacks rely on similar mechanisms like sniffing data from radio signals, injecting data into an existing connection or sending directly data to a target device, recovering cryptographic material, etc. Therefore, we propose to define a system in which these attacks can be described as a combination of a subset of elementary functions, thus providing us with a higher-level view of what they are, how they are carried and more importantly what elementary functions are involved for each of them. This would help shape a limited set of tools that can be used with a wide range of wireless protocols, factorizing some key parts of these attacks and therefore reducing the fragmentation.

If we can break down most wireless attacks into a succession of steps with some of them quite generic, we would be able to create a set of tools that mixes generic tools with protocol-specific tools that can collaborate to implement any wireless attack, and why not help set up new attacks on recent protocols.

1.3 State of the art

Most previous research focused on the design of relevant taxonomies to classify wireless attacks. As a result, various attack classifications have been proposed over time [15]. One of the most used is the active-passive classification, first introduced by Kent [16], splitting attacks in two categories in function of the presence or the absence of interactions with the target system. Similarly, the internal-external attack classification has been proposed by McNamara in [22], depending on the level of access to the network. While these taxonomies provide relevant information, the granularity is too coarse to express the technical differences between attacks. An alternative approach proposed by McHugh in [21] classify attacks based on the protocol layers they exploit, but leads to several limitations when heterogeneous protocols stacks are used or when attacks target several layers simultaneously. Finally, Stallings proposed [34] a classification based on the security services being compromised by attacks, splitting them into fabrication attacks (targeting authentication), interception attacks (targeting confidentiality), modification attacks (targeting integrity) or interruption attacks (targeting availability). This latest classification has been enriched in [18] to integrate domination attacks, targeting several security services at the same time.

Our literature review on wireless attacks leads us to identify two significant limitations. First, most papers have a limited scope: for example, a lot of papers only consider one single protocol, generally the Wi-Fi protocol (IEEE 802.11) [2, 37]. These studies classify the existing attacks

on Wi-Fi. Other studies consider Bluetooth Classic [35], or ZigBee [19], and provide a good classification of the existing attacks against these protocols.

Second, these papers are systematically based on classification: while they can be useful as they present a comprehensive list of attacks for a protocol and make it possible to identify similarities between attacks, they did not identify the atomic operations composing attacks nor their dependencies or relationships, contrarily to a *systematization*. We did not find any previous study breaking down attacks against wireless protocols and proposing a systematization of wireless attacks independently of any wireless protocol.

2 Systematization of attacks on wireless protocols

The first logical step in elaborating this systematization consists in defining a comprehensive threat model that will help us understand what an attacker may want to obtain by attacking a wireless network and the ways they can achieve their goal.

2.1 Considered protocols

In this paper, we mainly consider a set of well-known wireless protocols, with a focus on the ones used in the context of IoT. The considered protocols are introduced in Table 1.

Protocol	Modulation	Frequency band	Topologies	Use case
Bluetooth Low Energy	GFSK	2402-2480MHz	Peer to Peer	Small IoT devices
Bluetooth Mesh	GFSK	2402-2480MHz	Mesh	Small IoT devices
Enhanced ShockBurst	GFSK	2400-2499MHz	Star	Microcontrollers
Logitech Unifying	GFSK	2400-2499MHz	Star	Keyboards & mices
WiFi	OFDM	2412-2472MHz	Star	Computers & Smartphones
RF4CE	O-QPSK	2405-2480MHz	Peer to Peer	Remote Controls
ZigBee	O-QPSK	2405-2480MHz	Mesh	IoT devices
LoRaWAN	LoRa	433MHz/868MHz	Star	Sensors & Actuators
MeshTastic	LoRa	433MHz/868MHz	Mesh	Computers & Smartphones

Table 1. List of considered protocols.

2.2 Defining a generic Threat Model

Unfortunately, none of the wireless protocols studied defines a threat model, as a consequence we must define a generic one that may fit all of them. Therefore, we need to consider every possible outcome from the attacker's point of view first, and then define all the ways to achieve them considering wireless protocols as a whole. This first analysis is high-level, but it will give us the big picture of the expected goals and the various ways to achieve them.

Since wireless protocols are defined to make two or more devices exchange information, we consider by default any protocol as a *network protocol*, even if there is no real network defined by the protocol itself. A network can be identified by a specific *identifier*, be it defined in the protocol specification or tied to the device that manages the network, and is composed of two or more *nodes*. Node roles may vary depending on wireless protocols and their supported topologies, with some roles strongly tied to a specific protocol. The proposed systematization simply considers a node as a participant in a network regardless its role to define a generic threat model that fits the majority of wireless protocols. The resulting threat model has some limitations due to these considerations:

— one or more threats related to a specific protocol may not be considered as they may not fit other protocols
— role-based threats have not been included in this threat model and therefore will not be covered in the proposed systematization

We have identified the following potential objectives of attacks on a wireless network, each representing a threat that must be carefully addressed:

— identifying all nodes belonging to a network
— disrupting communications between two or more nodes
— joining a network as a legitimate node
— interacting with a node
— tampering with data sent between two nodes
— accessing sensitive information

2.3 Studying Well-Known Attacks

With these generic threats defined, we will now determine how they can be achieved based on the analysis of multiple wireless protocols we have previously studied: Wi-Fi, Bluetooth Low Energy, ZigBee, RF4CE, Logitech Unifying, Enhanced ShockBurst, and LoRaWAN.

Identify all nodes belonging to a network Identifying all the nodes that belong to a network is a classic step performed by attackers while targeting a wireless network (and it is quite the same for wired networks) to find potential targets. Wireless protocols like Wi-Fi, ZigBee or RF4CE have their nodes regularly sending data to other nodes on the network, or simply broadcasting some information to identify themselves in order for other nodes to connect to a network (known as *beaconing*).

An attacker can **passively capture data on a specific communication channel** where a node with a network *manager* role advertises itself while others send data, including their own identities. This approach allows the attacker to remain completely stealthily while identifying all the nodes within a network. This approach works pretty well for some protocols like Wi-Fi or ZigBee, because they generally leak their identities when sending data to other nodes of the network.

Wireless protocols like Wi-Fi, however, provide some features allowing a node in charge of the network to avoid advertising this network, forcing any node that needs to connect to query the network before accessing it through what the protocol calls *probe requests*. This is a method designed to actively search for a specific network, usually called an *active scan*.

An attacker can mimic this behavior and send data to discover active nodes, especially when most of them do not advertise themselves or send some data on a regular basis. This technique implies **sending multiple queries and listening for answers**.

Some protocols may also expose a subset or their entire topology, and discovering other nodes may be part of the protocol itself. An attacker simply has to **exploit any discovery feature available in a protocol to identify the nodes** that belong to a specific network.

In summary, an attacker can discover the nodes belonging to a network by:

— capturing passively data sent over a communication channel
— sending requests and capturing any response that follows this request
— taking advantage of a wireless protocol discovery mechanism

Disrupting communications between two or more nodes This threat is basically a *denial of service (DoS)* against a whole network or a part of it, and there are multiple ways to achieve this depending on the targeted network.

One way to disrupt communications is to simply jam the communication channel, the medium used to convey information between at least

two nodes of the network. Jamming consists of sending incoherent data on the communication channel, thus causing a lot of collisions if multiple nodes are sending data at the same time. Alternatively, communications can simply be disrupted by occupying the channel to prevent legitimate nodes from using it for data exchange.

For Wi-Fi and ZigBee networks, **simple jamming using a radio transmitter using the same modulation is enough to disrupt a network**, because such networks use a single communication channel.

For protocols using a channel hopping mechanism like Bluetooth Low Energy or Logitech Unifying, disrupting the communication channel may be more difficult as the attacker needs to know the parameters used by nodes to jump from one channel to another. The bandwidth used by most of these wireless protocols makes radio-based jamming quite difficult or at least very expensive as an attacker needs to effectively jam a wide frequency band. Some protocols rely on a very robust modulation combined with multiple channels, such as LoRaWAN, that makes jamming attacks more difficult to achieve (but still possible, see [13]).

When a wireless protocol implements a connection between two devices, like Bluetooth Low Energy or Wi-Fi, another way to disrupt communications between devices is to target an existing connection instead of the communication channel. In this case, the disruption targets only two devices but can be more effective and selective. **Causing a timeout in a connection using selective jamming** or terminating a connection through *data injection* are simple ways to break connections. This is possible for instance with Bluetooth Low Energy by injecting a *TERMINATE_PDU* as demonstrated in [8], or in Wi-Fi networks by injecting a deauthentication management frame as recalled in [37].

For wireless protocols that support a single connection like Bluetooth Low Energy, **maintaining a connection with a node** makes it disappear and avoids further connections from other nodes, thus leading to a denial of service.

Last, a fourth possibility has emerged from protocols that let nodes update connection parameters such as Bluetooth Low Energy (using a *Connection Parameters Update* procedure) or WiFi (through a *Channel Switch Annoucement*). This feature can be abused by an attacker to cause a desynchronization through data injection, making one of the two nodes participating in a connection change its parameters while the other sticks with the previous ones. *Selective jamming* can also be used to prevent one node from sending to the other the new communication parameters,

causing a desynchronization if the sending node does not expect any type of confirmation from the recipient.

As demonstrated above, disrupting communications can be achieved by:

— occupying the communication channel
— jamming selectively
— injecting data
— maintaining a connection

Joining a network as a legitimate node An attacker may want to join an existing network as a legitimate node, either by adding a node to the network or by spoofing an existing node. This could also include bypassing some security mechanisms required to join a network, such as breaking an encryption key or finding a password.

If we consider non-connected wireless protocols like Enhanced ShockBurst, joining a network can then be reduced to the **possibility of sending valid data to any node belonging to a network**. If encryption is used, the knowledge of the encryption key is required to send data that will be interpreted as legitimate.

For connection-based protocols like Wi-Fi or Bluetooth Low Energy, joining a network consists of **establishing a connection from an allowed node** and may require the knowledge of a password (or key) if the target network is protected. Regarding Bluetooth Low Energy, some connection modes may also require extra information about the network like an *identity Resolving Key*. All nodes may not be allowed to connect to a network, depending on its security settings: Wi-Fi can use a whitelist to only allow known nodes to establish a connection, as well as Logitech Unifying. For an attacker, that means **spoofing the identity of a legitimate node** may be required to be allowed to establish a connection. Let us note that depending on the protocol, it may be possible to have a limited interaction with a node without being able to properly initiate a connection (e.g., spoofing management frames in a WiFi network). We can assimilate this case to a frame injection in a non-connected wireless protocol, where only a subset of vulnerable frames can be injected.

Spoofing a node identity while sending valid encrypted data (implying the attacker knows the encryption key) can be assimilated to *data injection*, except if the protocol allows the co-existence of at least two connections from the same node but identified with two unique connection identifiers.

As a result, joining a network can be achieved by:

— sending valid data to a node (in plaintext or encrypted if the attacker knows the encryption key or password), optionally spoofing a node identity if required
— establishing a connection to a network from an allowed node, optionally spoofing a node identity if required

Interacting with a node For networks that do not require a connection, interacting with a node is generally simple as one has just to **send valid data and wait for an answer if it is expected**. Receiving data is done by listening on the communication channel and waiting for the answer to be sent by the target node.

For networks that do require a connection, it is pretty much the same except that a connection needs to be established first, as described in Section 2.3. The data exchanged with the node needs to be valid and may require the knowledge of an encryption key, or a password, if encryption is required by the protocol.

Interacting with a node may allow an attacker to access sensitive information if it is not protected or to query the node for information. If it is a smart lock for instance, an attacker may be able to send the correct data to make it open, thus implying a physical consequence. Replay attacks are also a good example of this kind of interaction, as an attacker only has to capture a legitimate interaction, even encrypted, and replay it to a target node.

Another possible way to interact with a device consists in injecting valid data into an existing connection or sending it directly to the target node. Doing so generally requires to be able to spoof the sending node identity.

An attacker can therefore interact with a node by:
— directly sending valid data to a node
— establishing a connection first and then sending valid data to a node
— injecting data

Tampering with data sent between two nodes Modifying data exchanged between nodes can be an excellent way to bypass security measures, capture sensitive data or trigger unspecified behaviors, and wireless protocols are generally prone to this attack. Wi-Fi is vulnerable to the *Evil twin* attack (see [37]), Bluetooth Low Energy to a man-in-the-middle attack through a spoofed node (see [6, 33]) as well as Logitech Unifying (see [23]).

Advanced attacks have also been developed like *injectaBLE* [8] which performs a **desynchronization of a connection between two nodes**, the attacker taking control of the connection by relaying any data sent by each desynchronized node to the other. This attack is totally transparent for the target nodes and does not require the use of a rogue node.

Man-in-the-middle attacks can be set up in different ways:

— a rogue node is created and other nodes are forced to connect to it and it will relay data to the intended target node
— a connection between two nodes is de-synchronized and the attacker relays all the data sent by each node to the other

Once the attacker has access to the data exchanged between two nodes, tampering data may be possible if the following conditions are met:

— data is not encrypted or the attacker knows the encryption algorithm and key if encryption is used
— data format is known from the attacker
— authentication and integrity fields (if any) can be modified to match the expected values

As a matter of fact, attackers rely on the following to tamper with data exchanged between two nodes:

— setting up a spoofed node that supports the target protocol and accepts connections if required by the protocol
— establishing a connection to the target node
— modifying data to achieve a specific goal
— sending data to a specific node, through an established connection or directly depending on the context
— receiving data from a connected node

Accessing sensitive information Wireless protocols are used to exchange information, including sensitive information. Sensitive information includes health data, personal information, passwords, encryption keys, etc. Access to this information should be protected against attackers.

On protocols that do not use encryption by default, such as Bluetooth Low Energy or Enhanced ShockBurst, accessing the information exchanged between two nodes is pretty straightforward and involves passive data capture. An attacker simply needs to **listen to the communication channel and capture all the information** sent on it. This can also be done on open Wi-Fi networks.

If encryption is used, the knowledge of the encryption key can be enough to decrypt the content of any communication except when session keys are used. In this case, it might be more difficult and sometimes

impossible to extract information. Wi-Fi protocol for instance was prone to a cryptographic attack on its *Wired Equivalent Privacy* (*WEP* in short) feature as it relies on a wrong usage of RC4 encryption, allowing attackers to perform a statistical attack on captured data and recover the encryption key (see Wikipedia [40] for more details on this attack). Let us note that if the encryption key is recovered, it may be used to decrypt past encrypted traffic if Perfect Forward Secrecy is not guaranteed by the protocol [3].

Information can be directly extracted from the data exchanged between nodes, but it is also possible when conditions are met to **recover extra information from a captured communication like encryption keys** and/or data if weak encryption is used. Moreover, it is sometimes possible to **infer specific information from captured data** like the *Generic Attribute* (or *GATT*) profile of a Bluetooth Low Energy target device by simply monitoring the services and characteristics discovery procedure.

An attacker can therefore capture sensitive data by:

— passively capturing data sent over a communication channel
— performing a man-in-the-middle attack against two nodes and capturing the data exchanged
— capturing and decrypting the exchanged data if encryption is used and the key/password is known or recovered
— recovering information using correlation and inference from the captured data

2.4 Identified attack primitives

The previous breakdown of various attacks on wireless protocols helped uncover a set of attack primitives used to realize the different threats described in our threat model. These primitives describe very basic operations that can be performed by attackers, and may be generic (independent of any protocol) or protocol-specific.

These primitives are defined to be as simple as possible and to be combined to create more complex attack scenarios, as those described in the previous section. Each primitive can be implemented in different ways but relies on the same basic operation, depending on a given protocol for protocol-specific primitives or generic in other cases.

They can be associated with three different layers based on the OSI model [39]:

— the *physical layer* or *PHY* corresponding to the layer closely associated with the physical connection between devices,
— the *logical link layer* corresponding to the protocol layer that transfers data between nodes on a *network segment*,

— the *network layer* corresponding to the layer allowing data transfer from a source to a destination through one or more networks or subnetworks

In wireless protocols, their own terminology will map more or less clearly to the OSI layers. To keep it simple, we will refer to the lowest layers as a *physical layer* (or *PHY*) and a *link layer*.

The diagrams representing these primitives in the following sections rely on a specific color code used to identify the flow of data and the type of operation that each primitive supports:

— primitives in blue process data received from a communication channel
— primitives in red send data destined to a communication channel
— primitives in purple receive data from and send data destined to a communication channel
— primitives in orange are generic primitives that can process data received from or destined to a communication channel

PDUs (*Protocol Data Units*) are blocks of data, including data exchanged between nodes but also additional metadata, that are manipulated by the primitives described below.

Primitives related to the physical layer This first attack primitive that happens at the physical layer consists in capturing data from a communication channel. This primitive is mostly protocol-specific, as each protocol has its own communication channel that may differ from others.

This primitive takes in input any parameter required to describe the communication channel and outputs captured data in the form of a series of *PDUs*. These *PDUs* can be used by another attack primitive or saved in a file for later use. This primitive is illustrated in Figure 1.

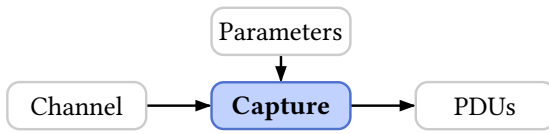

Fig. 1. Capture primitive

In opposition to the *Capture* primitive, the injection primitive allows an attacker to directly send data to a target node without caring about any connection. It takes *PDUs* in input and sends them into the communication

channel. This primitive does not spoof any node identity, it is simply used as an interface layer between the target network and other primitives.

This *Injection* primitive has no *spoofing* capability. Figure 2 shows the inputs and outputs of this primitive. Again, this primitive depends on the medium used by the associated wireless protocol.

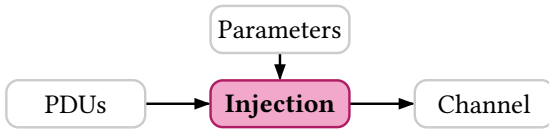

Fig. 2. Injection primitive

Primitives related to the link layer The *Connection* primitive is required by multiple attack scenarios, as an attacker may need to initiate a connection to a network as part of a greater attack. Initiating a connection is therefore a key attack primitive and is mostly protocol-dependent. The connection initiation procedure may be specific to the protocol used on the communication channel. It handles the state of a connection, is able to send and receive data and processes the received data regarding the wireless protocol it supports. In fact, this primitive can be described as a combination of *Capture* and *Injection* primitives with a specific layer handling the connection state, as shown in Figure 3.

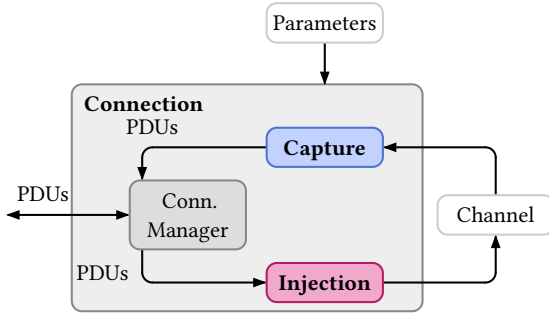

Fig. 3. Detailed Connection primitive

The connection primitive can be configured with any parameter required by the target protocol to initiate a valid connection to the network.

This may include any identifier of the target network as well as parameters related to the communication channel.

This primitive takes in input any parameter required to establish a valid connection to a network, and once this connection is established it can send and receive *PDUs* back and forth. Unlike the *Capture* primitive previously described, this primitive handles two flows:

— one flow taking data from the communication channel in input and issuing *PDUs* (in blue)
— another flow taking *PDUs* in input and producing data on the communication channel (in red)

The *Connection* primitive is used to establish a connection, but it could be interesting for an attacker, as previously mentioned, to synchronize with a target connection in order to capture any data exchanged or inject arbitrary data into it, targeting a specific node. This is done thanks to the synchronization primitive: injection is performed by spoofing one node or another and sending arbitrary data to the other node, while it captures any data exchanged over the target connection. Therefore, this *Synchronization* primitive also relies on a combination of *Capture* and *Injection* primitives, like the *Connection* primitive, as shown in Figure 4.

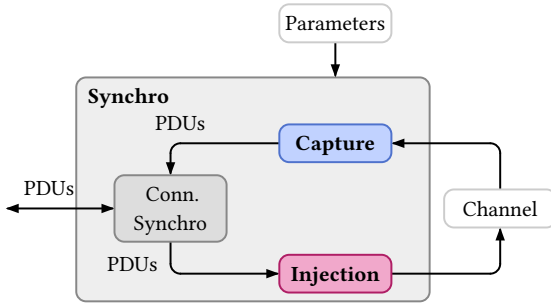

Fig. 4. Detailed Synchronization primitive

This primitive needs all the required parameters to synchronize with a target connection, at least the identifier for the target connection as well as some extra parameters required to correctly set the radio settings. Once synchronization is successfully done, captured data is sent as output and any incoming data is injected into the connection as if it was sent by a node specified in the parameters. This primitive handles two flows:

— one flow taking data from the communication channel in input and issuing *PDUs* (in blue)

— another flow taking *PDUs* in input and producing data on the communication channel (in red)

A third primitive, *Spoofing*, allows an attacker to emulate a specific node in a network or subnetwork, usually spoofing a specific node identity. Unsurprisingly, it also relies on the same low-level primitives as *Connection* and *Synchronization*, i.e *Capture* and *Injection*. Figure 5 describes a detailed model of this primitive, similarly to *Connection* and *Synchronization* primitives.

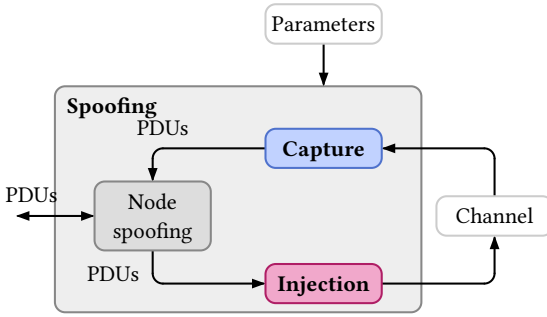

Fig. 5. Detailed Spoofing primitive

This primitive also defines two flows like the *Connection* primitive, but instead of initiating a connection it accepts connections from other nodes.

Note that *Connection*, *Synchronization* and *Spoofing* primitives are usually not able to directly send *PDUs* as they may need some time to establish or synchronize with an existing connection or receive a connection. These primitives keep *PDUs* in an internal queue until a connection is established or synchronized with and send them after.

A last primitive, *Jamming*, provides the ability to jam any communication by using two concepts described in [9] (section 7.2.2, p.157):

— **deceptive jamming:** a deceptive jammer continuously transmits packets on the targeted channel,

— **reactive jamming:** a reactive jammer only transmits an arbitrary radio signal when a packet transmission is detected on the targeted channel.

Like all previous link-layer primitives, *Jamming* relies on both *Capture* and *Injection* primitives and can be modeled as shown in Figure 6.

Primitives related to PDU manipulation Every physical layer or link layer primitive manipulates *protocol data units* (*PDUs*), which is the

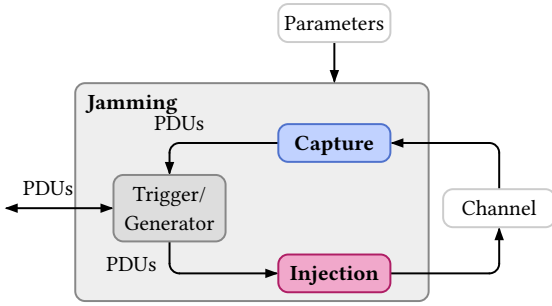

Fig. 6. Detailed Jamming primitive

simplest form of data they are able to convert into wireless frames to transmit, or to produce from a received wireless frame. Any operation above the link layer is therefore performed on *PDUs* related to a specific protocol, but some of them are somehow generic and can be defined as primitives used in attacks.

Moreover, these primitives do not exclusively rely on direct interaction with the communication medium and therefore can be used in both online and offline attacks, unlike the previously described primitives.

The first primitive of this kind is the *Extraction* one which allows an attacker to extract information directly from a single *PDU* (*selection*). The extracted information can then be used by an attacker to deduce information through *correlation* or *inference*. This primitive, illustrated in Figure 7 takes *PDUs* in input and gives the requested information in output. Parameters of this primitive include details about the information to extract.

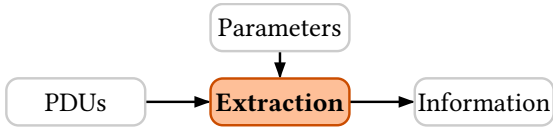

Fig. 7. Extraction primitive

The second primitive is about on-the-fly *PDU* modification and is called *Transform*. It applies a transformation to *PDUs* that match some requirements. The transformation and the associated requirements are provided in parameters in the form of a selection function and a transform.

This transformation can be used to tamper with any *PDU* content, modifying one or more information stored in a *PDU* as well as encrypting or decrypting content depending on its content and extra parameters provided to the primitive. This primitive takes *PDUs* in input, applies a transform on selected *PDUs* and outputs them, as described in Figure 8.

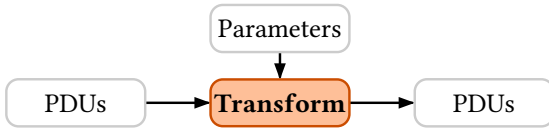

Fig. 8. Transform primitive

A third primitive called *Forge* provides the ability to craft specific *PDUs* in order to use them with other primitives, with their associated metadata, as described in Figure 9.

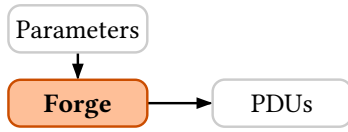

Fig. 9. Forge primitive

Another useful primitive is *Dump*, which provides the ability to save *PDUs* into a file in order to use them later. This is a very simple primitive that allows offline attacks on encryption for instance. This primitive takes in input a stream of *PDUs* and saves them into a file specified in its parameters, as illustrated in Figure 10.

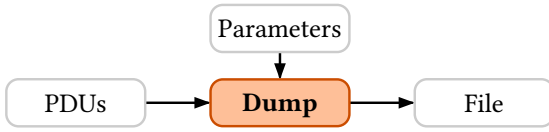

Fig. 10. Dump primitive

The last primitive, *Replay*, takes a specific file and loads *PDUs* from it, producing a stream of *PDUs* that can be used by other primitives. It is illustrated in Figure 11.

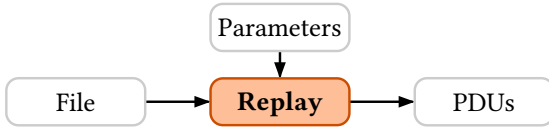

Fig. 11. Replay primitive

Summary of primitives dependencies The following Figure 12 summarizes all the primitives we identified that can be combined to describe any attack considered in this systematization, organized by their associated OSI layer.

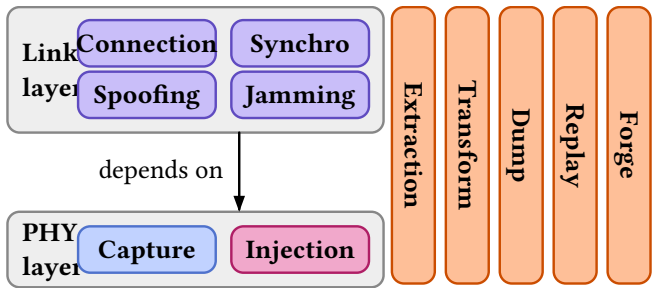

Fig. 12. Primitives hierarchy

Primitives associated with the link layer depend on those associated with the physical layer (*Capture* and *Injection*), while other primitives (*Extraction*, *Transform*, *Dump* and *Replay*) are not associated with a specific layer as they can be used on both of them.

2.5 Combining these attack primitives to model attacks

Attacks against wireless protocols can be defined as a combination of primitives that may imply a certain order in how the attack is carried out. Man-in-the-middle attacks for instance require that both *Connection* and *Spoofing* primitives are up and working to allow data processing, because if one of them is not then this attack will result in a denial of service. Setting up an attack can be seen as a *processing chain* that makes data flow from one primitive to another, but also as a series of steps that must be realized in a particular order.

Therefore, combining primitives is not only about creating this data processing chain, but also organizing them to reflect the successive steps used to perform the attack.

For clarity purpose, parameters of primitives used in the following diagrams are not shown except when necessary for proper understanding.

Identify all nodes belonging to a network Previously described in section 2.3, the discovery of nodes belonging to a network can be achieved in three different ways. The first way consists of simply capturing data from the communication channel and extracting any node information available. This passive attack is modeled in Figure 13. and relies on the *Capture* and *Extraction* primitives.

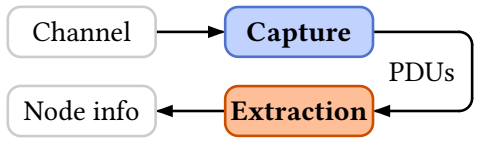

Fig. 13. Passive node identification

The second way to identify nodes is to interact with them in the network by injecting (*sending*) data that will cause these nodes to react, and capture the data sent by these nodes in order to identify them. This attack is modeled in Figure 14, and shows the two concurrent activities: one that relies on the *Injection* primitive that sends data into the network and another that captures data and extracts node information based on the *Capture* and *Extraction* primitives.

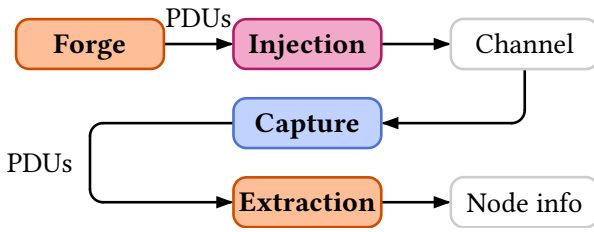

Fig. 14. Active node identification

The third and last option to identify nodes that belong to a network is to connect to the network and use a protocol feature to discover these nodes. This attack is modeled in Figure 15 and relies on a *Connection* primitive that will send one or more specific requests and extract information from the received *PDUs*.

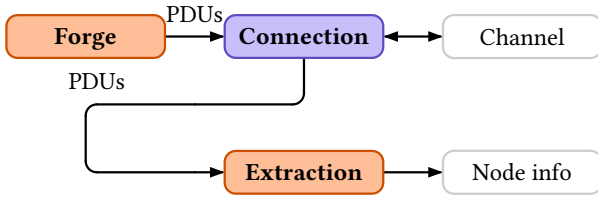

Fig. 15. Protocol-based active node identification

Disrupting communication between two nodes We have previously identified three different ways to disrupt communications between nodes, the first one being the simplest based on naive jamming, as described in Figure 16. This attack simply jams the communication channel.

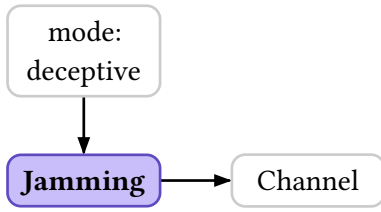

Fig. 16. Naive jamming

A second way to disrupt a communication between two nodes, when dealing with connected protocols, consists in desynchronizing a node by jamming the communication channel at a specific time, triggered by a specific pattern of data sent by one node to another, thus causing the connection to be considered as lost by one of the target nodes. This reactive jamming attack is modeled in Figure 17.

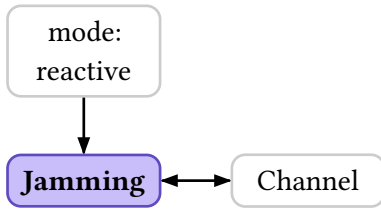

Fig. 17. Reactive jamming

A third way to disrupt communications is by injecting a specific PDU into a connection, spoofing a node identity. Figure 17 shows the model of this attack. Parameters passed to the *Synchronization* primitive contain:

— the identity of the node to spoof
— the parameters related to the communication channel to use

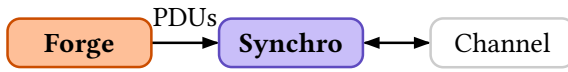

Fig. 18. Desynchronization

Joining a network When targeting a non-connected protocol, the simplest way to join a network is to send legitimate data into the communication channel, and process any received data. This is done thanks to an *Injection* primitive combined with a *Capture* primitive, as shown in Figure 19.

There is no connection to handle, it is based on sending data to and receiving data from different nodes that belong to a specific network. There is no connection to state management either.

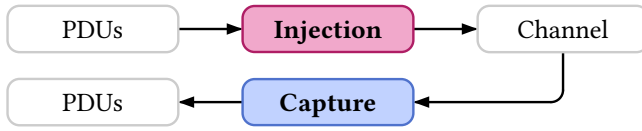

Fig. 19. Network join on non-connected protocol

Other protocols rely on connections between nodes requiring a specific procedure to be followed, that once established is identified by a unique identifier. In this case, we need to use a *Connection* primitive that implements a protocol connection process and allows exchanging data with other nodes once the connection is established. An attacker may spoof the identity of a legitimate node during this connection process, and this is also part of the *Connection* primitive (if fed with a specific node identity).

The connection state is handled by the *Connection* primitive itself. Figure 20 shows the model of this attack.

Replaying a captured communication Replay attacks are performed in two distinct steps. In a first step, the attacker captures a legitimate

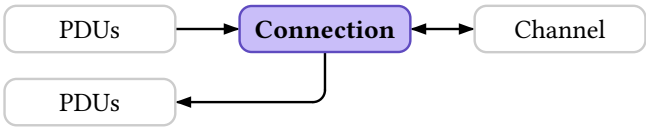

Fig. 20. Network join on connection-based protocol

communication between two nodes and saves it for later use. This captured communication is then replayed as-is and causes the target node to consider it legitimate (if vulnerable) and to react to it. Replay attacks can be performed at various layers: the physical layer for connection-less protocols or at the link layer for connection-based protocols.

In the case of a connection-less protocol, the attack relies on a *Capture* primitive to capture a communication and on a *Dump* primitive to save the captured data into a file. Then this file is later used to inject data into the communication channel with the combined use of a *Replay* and an *Injection* primitives, as shown in Figure 21.

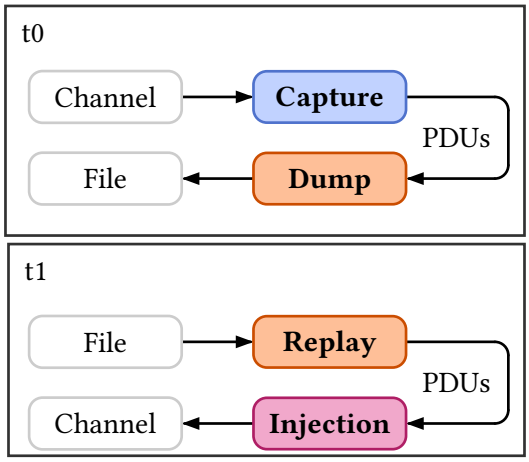

Fig. 21. Replay attack at the physical layer

For connection-based protocols, one simply needs to rely on corresponding link layer primitives, like *Synchro* and *Connect*. The attack is pretty much the same, but this time uses primitives that handle connections instead of blindly capturing data and sending it again, as demonstrated in Figure 22.

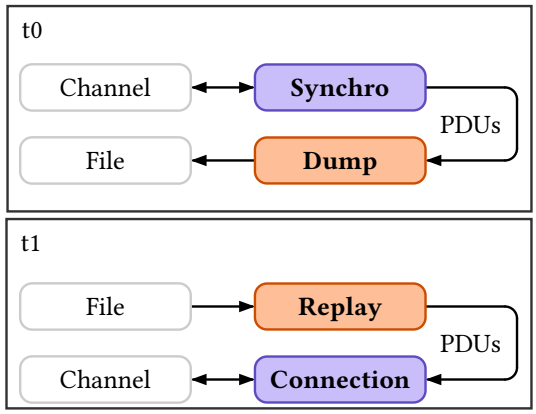

Fig. 22. Replay attack at the link layer

Tampering with data sent between two nodes Man-in-the-middle attacks are a pretty common way to tamper with a communication between two nodes, as described in section 2.3. These attacks are made possible by the possibility for an attacker to emulate a node that exposes the same identity as a legitimate targeted node, and force any other node to connect to his spoofed node while it relays data with the targeted node.

Two main scenarios are usually performed:

— in the first one, the attacker initiates a legitimate connection to the target node and then creates a spoofed node that mimics the legitimate one and then relays data back and forth (described in Figure 23)

— in the other, the attacker creates first a spoofed node mimicking a legitimate one and once a connection received on this rogue node initiates a connection to the target node and relays data back and forth (described in Figure 24)

If data is sent encrypted and the attacker knows the key and the algorithm used, it is possible to decrypt, modify and encrypt data in the *Transform* primitives used in the above attacks.

Accessing sensitive information The simplest attack aiming at capturing sensitive information is based on passive data capture, usually called *sniffing*. Sniffing data from a communication channel can be described pretty easily using our primitives, as shown in Figure 25. Attackers usually save captured data into files to avoid losing information, thus explaining the use of a *Dump* primitive in our model. It mostly works for connection-less protocols, not for connection-based protocols.

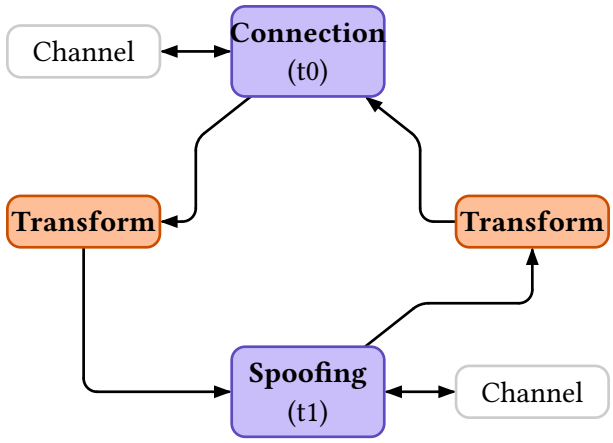

Fig. 23. Man-in-the-middle attack with on-the-fly data tampering (connect then spoof)

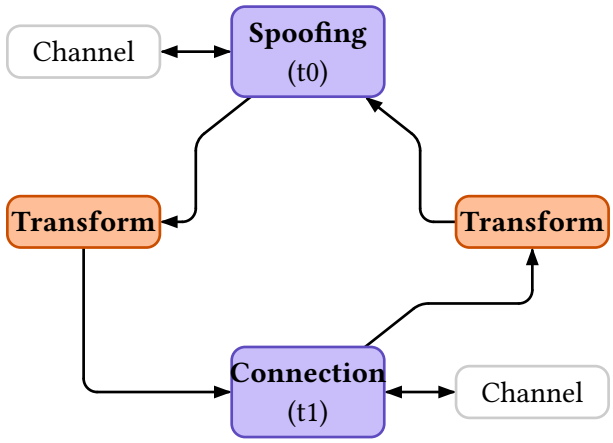

Fig. 24. Man-in-the-middle attack with on-the-fly data tampering (spoof then connect)

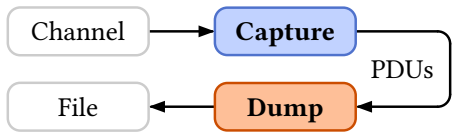

Fig. 25. Passive sniffing

When it comes to capturing data from a connection-based protocol, an attacker must first identify a connection and passively synchronizes with it. If the target protocol relies on a channel hopping mechanism, then synchronization will allow the attacker to recover the parameters required to hop from one channel to another and therefore capture the exchanged data. The attack model is slightly different, as shown in Figure 26.

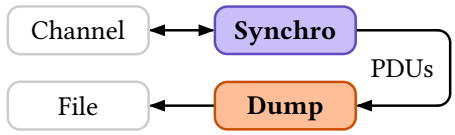

Fig. 26. Connection sniffing (synchronized)

Again, this is a good demonstration that attacks on connection-less protocols simply rely on primitives associated with the physical layer while connection-based protocols require the use of other primitives associated with the link layer.

If data is encrypted and the attacker knows the algorithm used and the key, then it is possible to use a *Transform* primitive to implement on-the-fly decryption and to capture decrypted data, as shown in Figure 27. The given example demonstrates on-the-fly *PDU* decryption related to a connection-based protocol, but the same is possible with a connection-less protocol by replacing the *Synchronization* primitive with a *Capture* primitive.

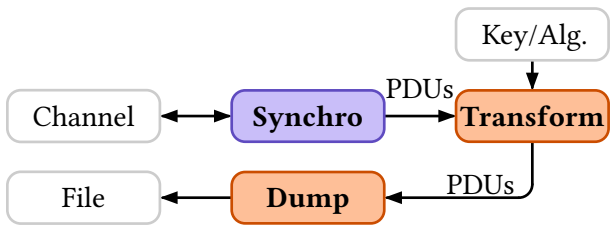

Fig. 27. Connection sniffing with on-the-fly decryption

Another way to extract sensitive information exchanged between two nodes is to perform a man-in-the-middle attack as previously described in section 2.5.

Besides accessing unencrypted information exchanged between nodes, an attacker can also collect data in order to break the cryptography used by a wireless protocol. These types of attacks are based on cryptographic weaknesses that require collecting relevant data from a protected network or a specific communication (e.g., vulnerable pairing). Once enough data is collected, an attacker can try to guess the encryption key used and decrypt the captured data. This attack is shown in Figure 28, and is quite similar to the one mentioned in section 2.3.

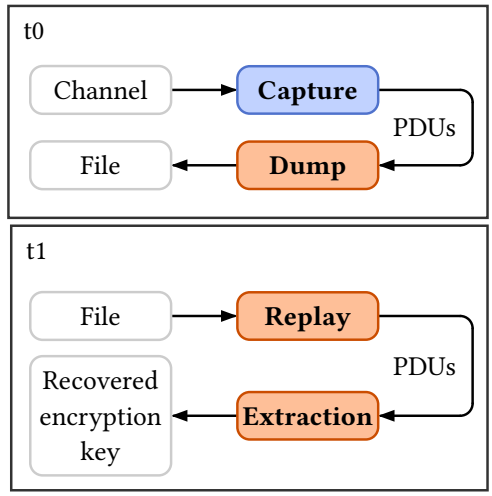

Fig. 28. Cryptographic attack

2.6 Limits of the proposed systematization

The attack primitives of this proposed systematization have been defined based on an analysis of common known attacks on wireless protocols, considering Physical Layer primitives at the *PDU* level only. While this choice limits the complexity of the systematization, it excludes RF-based attacks on the physical layer, that would require the definition of lowest level primitives. Defining such primitives would be a relevant extension to our work, increasing its expressiveness to represent low level attacks.

We also consider attacks targeting a wireless network topology, typically targeting protocols supporting mesh networks like *ZigBee* or *Bluetooth Mesh*, as out of the scope of this paper. While these attacks are not described in this paper, they can probably be described as a composition of our primitives.

These choices do limit the attacks that can be modeled with our primitives and therefore within this systematization.

3 Turning this systematization into standard and generic tools

The systematization of wireless attacks defined in the previous section can be seen as a way to define a set of tools that can be combined to carry out attacks, each tool corresponding to an attack primitive. This approach is quite similar to the *KISS* principle [38] that demonstrated its effectiveness in multiple implementations, one of the most known being UNIX. Using this KISS principle, a set of basic tools can be defined that once combined allow complex operations to be implemented, without the need to create a new tool for each complex attack.

3.1 Deriving tools from primitives

The eleven attack primitives defined in the previous section correspond to a limited and necessary set of tools that must be implemented to cover all the identified attacks on wireless protocols if there is a way to easily combine them to match the defined attack models. Some attack primitives must be implemented specifically for each wireless protocol while others are more generic and can be used no matter the protocol.

The *Capture*, *Injection*, *Connection*, *Synchronization*, *Spoofing* and *Jamming* primitives are strongly tied to the wireless protocol and its communication channel specificities, thus making non-generic attack primitives: they must be implemented for each targeted wireless protocol, resulting in as many variants as they are protocols to support.

The *Dump*, *Replay*, *Extraction* and *Transform* attack primitives however are generic as they directly manipulate *PDUs* with no care of the wireless protocol used: they give the user the power to save data sent or received through a wireless protocol, replay this data, but also extract any part of it or transform it the way the user needs. This genericity is possible thanks to the fact that *PDUs* are composed of a bit-stream, generally interpreted as a series of bytes, that models the data exchanged over a wireless protocol. Generic attack primitives result in the corresponding generic tools and can be used independently of the wireless protocol considered.

3.2 Combining tools to implement attacks

If we consider each attack primitive as a tool, we also need to define a way to combine them practically. Relying on the *KISS* principle is one thing, but implementing it another that may seem harder. As previously stated, tools can be combined to create a data *processing chain* (see 2.5) but may also rely on specific timing to work as expected. These two aspects must be taken into account when designing a way to combine these tools together.

The intuitive way that comes to mind is to implement each attack primitive as a command-line tool and to use UNIX-like systems chaining capability to chain them in order to follow an attack model. Following this principle, the passive node scanning approach described in section 2.5 can be designed to work as illustrated in Listing 1.

Listing 1: Combining tools in shell

```
1 $ capture [parameters] | extraction [parameters]
```

In this case, the *Capture* primitive can be configured through its set of parameters and chained to the *Extraction* primitive. Data flowing from the first primitive can then be transferred to the next one, and the *Extraction* primitive extracts the expected information from the data it receives.

This chaining benefits from UNIX-like shell's piping feature in a way it can be used to reproduce an attack model using command-line tools. But this way of chaining attack primitives has its own restrictions:

— UNIX-like pipe feature is a single-way communication mechanism, thus allowing a tool to uniquely send data to another tool but not receive data from it

— it defines both a specific timing and processing chain as the first tool is expected to output data to the second one in the chain, introducing a time constraint

These restrictions definitely limit the attacks on wireless protocols that can be implemented using command-line tools, but offer a quite simple way to experiment some of them using simple command-line tools.

Another approach that may better fit this systematization is the one used in *GNURadio* [14]: a graph of blocks representing an attack with each block representing a primitive (Figure 29 shows an example *GNURadio* flowgraph). These primitives can then be chained together using links, and timing may be part of the graph configuration.

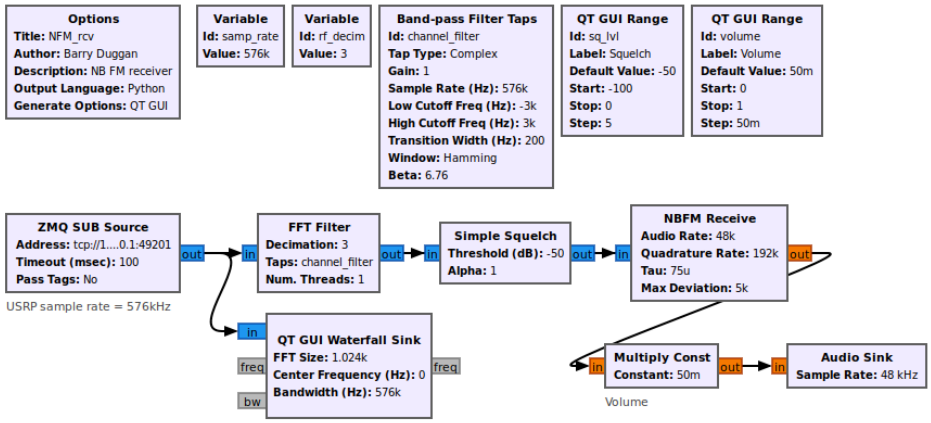

Fig. 29. Example of a GNURadio flowgraph to demodulate narrow-band FM

This approach offers greater flexibility but relies on a graph that needs to be designed either visually using a specific tool like *GNURadio* or programmatically using Python code.

Eventually, we opted for the command-line tool chaining instead of GNURadio's flow graphs because of its simplicity and ease of use.

3.3 Experimenting with WHAD

One of the goals of this systematization of attacks on wireless protocols was to determine which tools must be implemented in our *WHAD* framework [30], designed to provide an easy interface to wireless protocols as well as an easy way to implement and test attacks against a large range of wireless protocols used in the *Internet of Things* field.

Combining tools with pipes and sockets We previously demonstrated that UNIX-like shell pipes can be used to chain tools in a terminal and that this solution has some limitations, one of them being that this pipe mechanism is a one-way communication channel. Since we needed to experiment with two-way communication channels, we designed a way to use pipes to create such a communication channel between tools, similar to how *Mirage* uses the same mechanism to combine multiple modules [10]. Moreover, relying on shell pipes also allows third-party tools to interact with WHAD's and encourage future contributions.

Figure 30 shows the solution we designed in WHAD to combine attack primitives while allowing two-way communication. The principle is pretty simple: each tool that has its output piped to another tool creates a socket

server and sends to the piped tool, through its standard output, all the required information about this socket server. Piped tools can then connect to the corresponding socket server, creating a bi-directional chain of socket clients and servers from the first tool of the chain to the last one. This solution is a workaround designed to transform pipes into bidirectional communication channels.

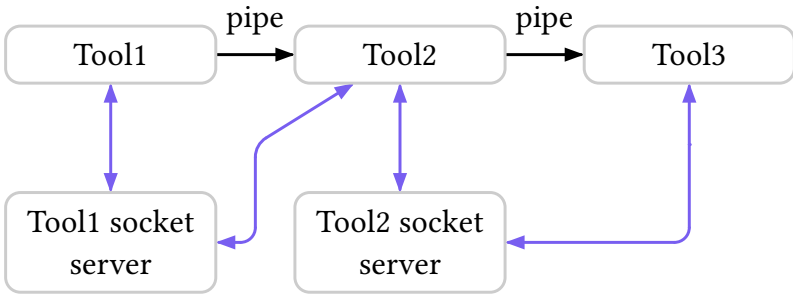

Fig. 30. Two-way communication channel

Generic primitives Four generic attack primitives have been defined in our systematization, and therefore needed to be implemented in WHAD. Since all these primitives manipulate *PDUs*, we needed an easy and efficient way to analyze and create *PDUs* in order to provide the expected features, and this is where *Scapy* [32] solves everything. *Scapy* already supports a lot of wireless protocols and provides easy packet dissection and assembly, all that is needed by our generic tools.

Four command-line tools have been implemented, following the guidelines defined in our systematization, supporting the above-mentioned chaining mechanism. *Dump* primitive has been implemented in *wdump*, *Replay* primitive in *wplay*, *Extraction* in *wextract* and *Transform* in *wfilter*. These last two tools accept user-provided Python code using *Scapy* to implement data extraction and on-the-fly *PDUs* modification.

Finally, we also defined an additional command-line tool named *wanalyze*, aiming to complement *wextract* with protocol-specific inference. While remaining a generic tool from an user perspective, it allows to implement a set of common inference algorithms taking advantage of known protocol weaknesses or specificities. This tool relies on *Traffic Analyzers*, which are basic algorithms taking PDUs streams as input and generating a set of named inferred data as output. *Traffic Analyzers* are implemented as a

state machine with three states, as illustrated in Figure 31: the **triggered** state is reached when a specific condition is met (for example, the first packet of a given procedure has been processed) and the final state **completed** is reached when another condition has been met (all necessary fields have been extracted and the inference has been successfully applied). When the state machine reaches the state **completed**, the inferred data can be collected. At any time, the state can be reset, leading to the **idle** state.

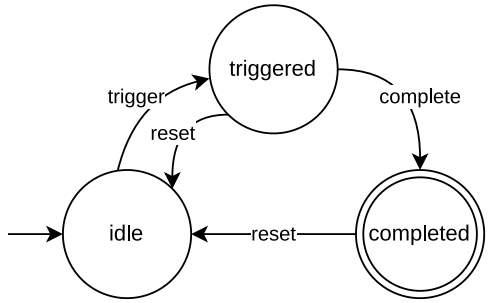

Fig. 31. Simplified state machine of a Traffic Analyzer

For every protocol, WHAD includes a set of specialized Traffic Analyzers, that can be easily applied on a given PDUs stream using *wanalyze*. The generated data can be applicative data such as keystrokes, audio stream or application profiles, as well as encryption key if a vulnerable pairing is targeted. For example, Bluetooth Low Energy, ZigBee Touchlink, RF4CE and Logitech Unifying protocols are all vulnerable to attacks targeting cryptography during their pairing phase, as demonstrated on Logitech Unifying and RF4CE in section 3.4.

Physical layer related primitives WHAD provides a specific command-line tool called *wsniff* that implements the *Capture* primitive (associated to the physical layer) for different supported protocols. This is not a generic tool per-se, but it concentrates in one tool all the different flavors of the *Capture* primitive for all the supported protocols. *wsniff* can be combined with *wextract* to capture *BLE* advertisements on specific channels and extract information from them, like the advertiser *Bluetooth Device Address*. The example below listens for advertisements on channel 37 and then extracts the advertiser address and the received signal strength indicator (*RSSI*) and outputs these values separated with a comma:

```
$ wsniff -i uart0 ble -c 37 -a | wextract -d ','
↪  "p[BTLE_ADV_IND].AdvA" "p.metadata.rssi"
a4:c1:38:60:fc:5c,-53
d0:d0:03:77:53:28,-65
a4:c1:38:60:fc:5c,-54
d0:d0:03:77:53:28,-65
```

This is an implementation of the attack described in section 2.5, but since it simply captures data from the communication channel and extracts parts of it, there is neither de-duplication nor post-processing.

The *Injection* primitive has been implemented in *winject*, a tool that supports injection for all of the supported protocols. *winject* can be combined with *wplay* to perform replay attacks that targets a 433 MHz wireless doorbell for instance:

```
$ wsniff -i uart0 phy --ask -f 433920000 -d 10000 | wdump
↪  doorbell.pcap
$ wplay doorbell.pcap | winject -i uart0 phy --ask -f 433920000 -d
↪  10000
```

In this example, we demodulate an amplitude-shift keyed signal and save the demodulated data in a PCAP file. This PCAP file can then be replayed at will using *wplay* combined with *winject*, the hardware handling the modulation. This is the exact attack described in section 2.5.

Link layer primitives Some additional command-line tools like *wble-connect* and *wble-spawn* implement respectively the *Connection* and *Spoofing* primitives for Bluetooth Low Energy, and can be combined to create a man-in-the-middle attack like described in section 2.5, following two scenarios.

The first one consists of first connecting to the target device and then spoofing its identity:

```
$ wble-connect -i hci0 a4:c1:38:60:fc:5c | wble-spawn -i hci1 -p
↪  device.json
```

In this specific case, we are using a Bluetooth Low Energy HCI adapter to first connect to the target device identified by the address *a4:c1:38:60:fc:5c* and then spoof the target device identity (saved in *device.json*) using another compatible Bluetooth Low Energy HCI adapter. *PDUs* received by *wble-connect* are directly forwarded to *wble-spawn*, and the same from *wble-spawn* to *wble-connect*. This is totally transparent to

the user. This example implements the same approach used by *Btlejuice* to perform a man-in-the-middle attack.

The second scenario is a variant of the first one. Instead of first connecting to the target device and then advertising a spoofed one, we are going to first advertise a spoofed device and once a connection is received we will connect to the target device:

```
$ wble-spawn -i hci1 -p device.json | wble-connect -i hci0
↪  a4:c1:38:60:fc:5c
```

These two examples show how the temporality of an attack can be defined using command line and pipes, especially with tools waiting for a specific event to happen (in this case a connection).

It is also possible to use *wfilter*, which implements two *Transform* primitives (one for the upstream traffic and one for downstream), in the processing chain in order to replace on-the-fly a value returned by a GATT server during a read operation for instance:

```
$ wble-connect -i hci0 a4:c1:38:60:fc:5c|wfilter --down -f -t
↪  "p[ATT_Read_Response].value=b'virtualabs'" "ATT_Read_Response
↪  in p and p[ATT_Read_Response].value == b'ESMLm_c9i\x00'" |
↪  wble-spawn -i hci1 -p lightbulb.json
```

Figure 32 shows the original device name characteristic value, this value is successfully changed through the man-in-the-middle attack described above as shown in Figure 33. Bluetooth addresses are different because no address spoofing has been set for this attack.

3.4 From atomic primitives to complex attacks

By leveraging and combining the implemented atomic primitives, it becomes straightforward to combine simple actions to perform more complex attacks. In this section, we describe the attack workflow illustrated in Figure 34 on two main wireless protocols presenting similar weaknesses, *RF4CE* (802.15.4-based protocol for Remote Control) and *Logitech Unifying* (used by Logitech wireless mice and keyboards).

Experimental setup We consider two wireless communications, respectively using *RF4CE* between a Remote Control and a reception dongle and Logitech Unifying between a wireless keyboard and its reception dongle. Both communications include two phases:

— First, a pairing phase is performed over the air to generate a shared encryption key.

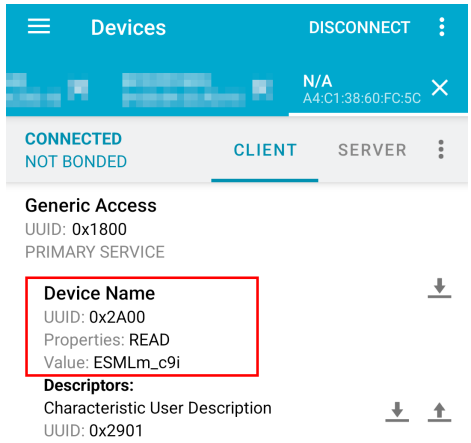

Fig. 32. Device name characteristic exposed by the device

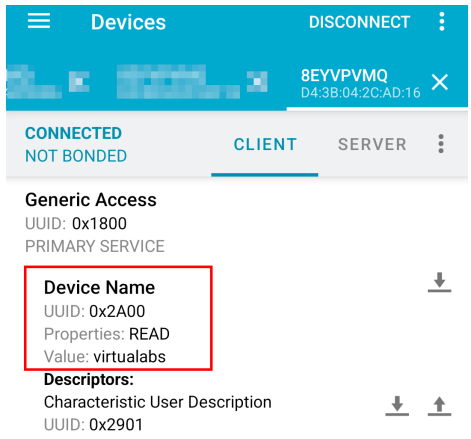

Fig. 33. Device name characteristic changed through MitM

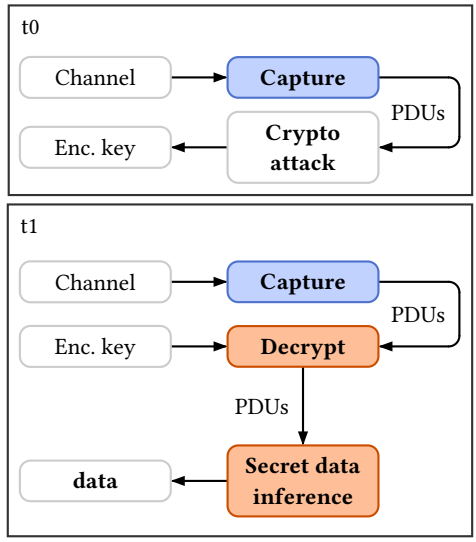

Fig. 34. Complex attack workflow inferring sensitive data

— Second, an encrypted session is established and sensitive information is transmitted using encrypted traffic.

The attacker goal is to gain access to the sensitive information transmitted during this second encrypted phase. In the case of Logitech Unifying, the attacker wants to infer the keystrokes transmitted between the wireless keyboard and the reception dongle, while in the case of *RF4CE* his goal is to gain access to the keypresses and the audio stream used for vocal commands. To do so, he will passively collect these communications, break the encryption key, decrypt the collected traffic and apply inference algorithms to recover the sensitive data.

Connection sniffing In order to passively eavesdrop the two phases of the communication and dump the *PDUs* into a PCAP file, the attacker must adapt its strategy to the targeted protocol, as described in section 2.5.

In the case of *RF4CE*, all the communication (including the pairing phase) occurs on a single channel, allowing to directly capture the communication similarly to a connection-less protocol as illustrated in Figure 25. Thus, once the channel in use has been identified, capturing the traffic with *wsniff* and dumping it into a capture file with *wdump* is straightforward:

```
1 $ wsniff -i uart0 rf4ce -c 15 | wdump rf4ce_comm.pcap
```

In contrast, *Logitech Unifying* makes use of two connection-oriented communications based on two distinct channel hopping algorithms:

— During the pairing phase, a specific given address (e.g., *BB:0A:DC:A5:75*) is used to distinguish the pairing from other communications. A fast channel hopping algorithm is specifically used for pairing.

— During the rest of the communication, the address negotiated during the pairing phase will be used to identify the communication. A lazy channel hopping algorithm will be used, where the communication remains on a single channel until a packet loss occurs.

These two phases must be captured independently using the strategy illustrated in Figure 26, since they require two different kind of synchronization (so-called *"–pairing"/"-p"* and *"–follow"/"-f"*):

```
$ wsniff -i uart0 unifying -f A8:41:9E:B5:0F | wdump
↪  unifying_comm.pcap
```

Breaking vulnerable pairings While Logitech Unifying and RF4CE rely on state-of-the-art encryption schemes (AES), they both present serious weaknesses for their pairing phase. Indeed, both of them rely on security by obscurity to derive the encryption key, applying basic deterministic transformations (e.g., XOR, position shifting) on values transmitted over the air in plaintext during the pairing phase. Both the involved packets and the two derivation algorithms are detailed in Figure 35 and Figure 36. As a result, once the derivation scheme has been reverse-engineered and disclosed publicly, inferring the encryption key from a capture of the pairing traffic is as trivial as implementing the algorithm. These attacks are thus similar to the cryptography attack presented in Figure 28. In WHAD, such implementations have been included as *Traffic Analyzers* (respectively *"key_cracking"* for RF4CE and *"pairing_cracking"* for Unifying) and can be executed using *wanalyze* command-line tool. Considering that an initial pairing phase has been captured in a PCAP file, retrieving the corresponding encryption key can be performed:

```
$ wplay rf4ce_comm.pcap | wanalyze key_cracking
[v] key_cracking → completed
  - key:  48ca7e9fdbc168b0297dd97d4f7f85a8

$ wplay unifying_pairing.pcap | wanalyze pairing_cracking
[v] pairing_cracking → completed
  - key:  02bea8b5ef61037e87882e4daebf403b
```

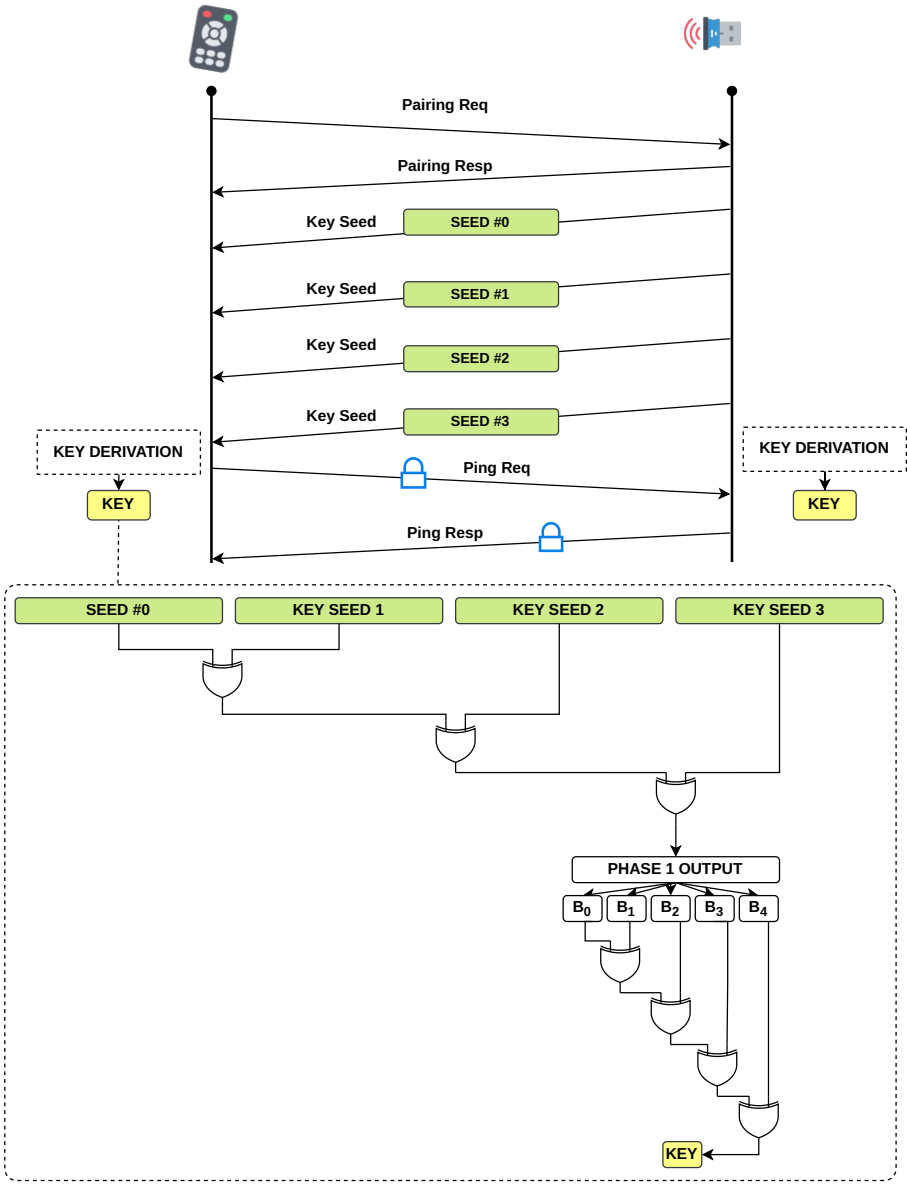

Fig. 35. Weak pairing phases of RF4CE

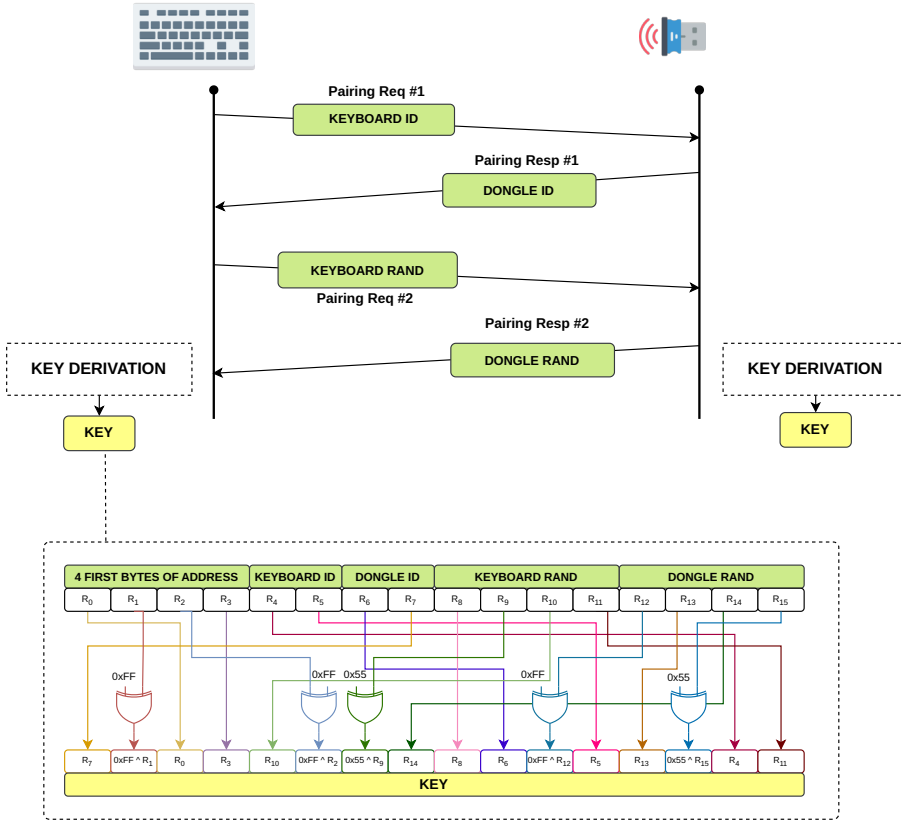

Fig. 36. Weak pairing phases of Logitech Unifying

Inferring sensitive data Once known, we can easily leverage this knowledge of the encryption key and replay and decrypt the traffic (*–decrypt/-d* option) by providing the key (*–key/-k* option) to the *wplay* tool. In a spirit of simplicity, we implemented the *Transform* primitive allowing to decrypt traffic to the *wplay* tool directly. Thus, performing an attack similar to the one presented in Figure 27 can be done easily using *wplay* only:

```
$ wplay rf4ce_comm.pcap -d -k 48ca7e9fdbc168b0297dd97d4f7f85a8
$ wplay unifying_comm.pcap -d -k 02bea8b5ef61037e87882e4daebf403b
```

Then, we can provide the decrypted traffic to various *traffic analyzers* in order to infer sensitive data from applicative layers. These analyzers can be more or less complicated, depending on the way the data are encoded. For example, *Logitech Unifying* keyboards transmit keystrokes as two consecutive HID scan codes, matching the Human-Interface Device specification. We implemented the HID mapping as a generic component since it is intensively used by multiple protocols (e.g., BLE, Unifying), and use this component to parse the keystrokes in the Unifying *keystroke* traffic analyzer. Extracting the keystrokes becomes trivial using *wanalyze* on the decrypted traffic:

```
$ wplay --flush unifying_comm.pcap -k
↪  02bea8b5ef61037e87882e4daebf403b -d | wanalyze keystroke
[v] keystroke → completed
  - key:  a

[v] keystroke → completed
  - key:  b

[v] keystroke → completed
  - key:  c
```

The *RF4CE* key presses transmitted by the remote control are encoded using a similar mechanism, relying on the mapping indicated in the Zigbee Remote Control Application profile. This mapping is also implemented in a traffic analyzer named *keystroke*:

```
$ wplay rf4ce_comm.pcap -k 48ca7e9fdbc168b0297dd97d4f7f85a8 -d |
↪  wanalyze keystroke
[v] keystroke → completed
  - key:  7

[v] keystroke → completed
  - key:  0

[v] keystroke → completed
  - key:  [TV POWER]
```

The Remote Control also supports a specific feature allowing to transmit audio stream over *RF4CE* encrypted traffic. Such feature is dedicated to the transmission of short vocal commands, triggered by a specific button press. When this button is pressed, the microphone is enabled, audio samples are then processed by the microcontroller and encoded using Adaptive Differential Pulse Code Modulation (ADPCM). Once encoded, the stream is split into blocks of 80 bytes and transmitted over the air using a specific *"Data Notify"* payload. Additional control payloads are also used to indicate the stream parameters (e.g., sample rate, resolution or channel number), its start and its end. Once reverse engineered and identified, this algorithm has been implemented in WHAD as a traffic analyzer named audio, allowing to infer the parameters and the samples from the corresponding packets and to decode the audio stream into a WAV file. It is thus possible to extract this audio stream from the decrypted PCAP file by combining *wplay* and the *audio.raw_audio traffice analyzer* as shown below:

```
$ wplay rf4ce_comm.pcap -k 48ca7e9fdbc168b0297dd97d4f7f85a8 -d |
↪  wanalyze --raw audio.raw_audio > stream.wav
```

3.5 The price of modularity

Implementing a set of tools based on these eleven attack primitives worked quite well and gave pretty good results as shown in the previous section, but it comes at a cost.

First, combining tools in a single command line may seem straightforward but it is quite the contrary: bi-directional communication between two programs chained in a single command line is not a standard feature. We managed to overcome this limitation by using both shell pipes and Unix sockets, but this implementation brought a lot of complexity.

Second, we had to create a lot more separate tools to carry out specific attacks or at least to provide the user with every possible primitive for

each supported protocol. By trying to reduce fragmentation amongst wireless attack tools, we fragmented our tools to allow flexibility and that could be seen as a failure. We limited this fragmentation in WHAD by implementing in a single tool a primitive shared by multiple protocols (like the *Capture* primitive implemented in *wsniff*), but again with a cost: this tool has a lot of options and is quite complex to use, while we wanted to have simple tools to combine.

However, the benefits of modularity are far higher than the cost of designing and debugging such tools. It is definitely more challenging to implement a tool that can be combined with other tools using the mechanisms described in this paper, but it is more satisfying to be able to draft an attack and to be able to try it directly by combining existing tools rather than developing a new tool to test this attack. We have been amazed many times how effortlessly we could experiment with new attack ideas within minutes. This was achieved without the need to develop new tools, simply by combining various attack primitives using our dedicated tools.

3.6 Designing tools for other wireless protocols

Another effective demonstration of this modular system's efficiency is its ability to seamlessly integrate with other wireless protocols with minimal effort, particularly when these protocols are built upon a supported low-level protocol or modulation. We recently stumbled upon the *Meshtastic* protocol [24] that relies on *LoRa* modulation to provide a low-power wide-range communication system using an open-source protocol and software. *LoRa* being natively supported by WHAD, this protocol was a good fit to test the framework capabilities and to put our basic command-line tools to the test with a real-world case.

Low-level primitives for exploration *wsniff* provides the *Capture* primitive for *LoRa* and was used to capture low-level *Meshtastic* data frames using the correct configuration. The required configuration was found by combining information from Meshtastic's frequency slot calculator [25] and firmware source code [26], and we then were able to capture data from a *Meshtastic* frequency slot (in this case using the *LONG-FAST* preset with the European 868MHz ISM band):

```
$ wsniff -i uart0 phy --lora -f 869525000 -crc -em
↪ --preamble-length 8 -sf 11 -cr 45 -bw 250000 -w 28
[ raw=False, timestamp=72863357, rssi=59, frequency=869525000,
↪ iq=[], endianness=LITTLE, deviation=0, datarate=0,
↪ modulation=LORA, syncword=0000 ]
<Phy_Packet  data=ffffffff30ffca06... |>

[ raw=False, timestamp=80452306, rssi=59, frequency=869525000,
↪ iq=[], endianness=LITTLE, deviation=0, datarate=0,
↪ modulation=LORA, syncword=0000 ]
<Phy_Packet  data=ffffffff30ffca06... |>
```

Captured data frames are wrapped into a *Phy_Packet* Scapy-based class, and they can be saved into a PCAP file for further analysis using *wdump*:

```
$ wsniff -i uart0 phy --lora -f 869525000 -crc -em
↪ --preamble-length 8 -sf 11 -cr 45 -bw 250000 -w 28 | wdump
↪ meshtastic.pcap
```

Custom tool development As mentioned in section 3.3, WHAD processing chain relies on *Scapy*'s packet definition. It is therefore easy to define a *Meshtastic* packet format based on the protocol documentation, using Scapy's field definitions (see Listing 2). This packet definition can be given to *wextract* to extract every sender node address:

```
$ wsniff -i uart0 phy --lora -f 869525000 -crc -em
↪ --preamble-length 8 -sf 11 -cr 45 -bw 250000 -w 28 | wextract
↪ -l meshtastic_defs "'0x%08x' %
↪ MeshtasticHdr(bytes(p)).sender_addr"
0x06caff30
0x06caff30
0x06caff30
```

Using WHAD's user-defined transform features, it is also easy to create a new command-line application that can be chained with *wsniff* to decrypt *Meshtastic* packets based on the above Scapy definition (Listing 2) and some cryptography routines, as illustrated in Listing 3.

Listing 2: Meshtastic packet definition

```
class MeshtasticHdr(Packet):
    """Meshtastic header:"""
    name = "MeshtasticHdr "
    fields_desc=[
        XLEIntField("dest_addr", 0xffffffff),
        XLEIntField("sender_addr", 0xffffffff),
        XLEIntField("packet_id", 0),
        BitField("hop_limit", 3, 3),
        BitField("want_ack", 0, 1),
        BitField("via_mqtt", 0, 1),
        BitField("hop_start", 0, 3),
        ByteField("channel_hash", 0),
        ShortField("rfu", 0),
    ]
```

This user-defined command-line tool can then be chained with *wsniff* to decrypt packets coming from a specific *Meshtastic* channel, as shown below:

```
$ wsniff -i uart0 phy --lora -f 869525000 -crc -em
↪ --preamble-length 8 -sf 11 -cr 45 -bw 250000 -w 28 | python3
↪ meshtastic_decrypt.py
###[ MeshtasticHdr  ]###
  dest_addr = 0xffffffff
  sender_addr= 0x6caff30
  packet_id = 0x2a72101e
  hop_limit = 3
  want_ack  = 0
  via_mqtt  = 0
  hop_start = 3
  channel_hash= 188
  rfu       = 0
###[ Raw ]###
     load      = '\x08\x01\x12\x0eThis is a test'
```

It is also possible to save in a PCAP file the decrypted *Meshtastic* frames using *wdump*:

```
$ wsniff -i uart0 phy --lora -f 869525000 -crc -em
↪ --preamble-length 8 -sf 11 -cr 45 -bw 250000 -w 28 | python3
↪ meshtastic_decrypt.py | wdump meshtastic.decrypted.pcap
```

Listing 3: meshtastic_decrypt.py

```
"""User-defined transform/extract script"""
import binascii
from struct import pack
from scapy.packet import Packet, Raw
from scapy.fields import XLEIntField, BitField, \
  ByteField, ShortField
from Crypto.Cipher import AES
from Crypto.Util import Counter
from whad.tools.user import user_transform
from meshtastic_layer import MeshtasticHdr

KEY = bytes.fromhex(('77d3f772c7bacf97b4df0f74c832a00d00a8bbf00dc
↪  0d332d899bd0f850b1f99'))

def encrypt_data(sender_addr, packet_id, key, data):
    """Encrypt a Meshtastic payload"""
    nonce = pack("<QQ", packet_id, sender_addr)
    ctr = Counter.new(128, little_endian=False,
      initial_value=int(binascii.hexlify(nonce), 16)
    )
    cipher = AES.new(key=key, mode=AES.MODE_CTR, counter=ctr)
    return cipher.encrypt(data)

def decrypt_data(sender_addr, packet_id, key, data):
    """Decrypt a Meshtastic payload"""
    return encrypt_data(sender_addr, packet_id, key, data)

def ingress(packet: Packet):
    """Process inbound packet"""
    return packet

def outgress(packet: Packet):
    """Process outbound packet"""
    # Parse Meshtastic packet
    if len(bytes(packet)) >= 16:
        packet = MeshtasticHdr(bytes(packet))
        # Try to decrypt packet
        packet.payload = Raw(decrypt_data(
            packet.sender_addr, packet.packet_id,
            KEY,bytes(packet.payload)))
        packet.show()
    # Forward packet
    return packet

if __name__ == "__main__":
    user_transform(outgress, ingress)
```

Modularity lets users focus on the essential We did not have to bother with how data is received or saved into a PCAP file since these tasks are already provided by existing tools based on known primitives (*Capture* and *Dump* in this case), we only had to focus on the essential tasks that were parsing the received data as *Meshtastic* packets, extracting interesting information and in the end decrypting data using a known key and algorithm.

Meshtastic being a quite complex protocol, we decided not to use *wfilter* to decrypt data but to create our own tool with the help of WHAD's features. We were able to implement our own *Transform* primitive for *Meshtastic* that decrypts received data and uses it to save decrypted data frames into a PCAP file for later analysis.

Moreover, the tool we created can then be used in other processing chains for other purposes, which will make further attacks or analyses easier.

3.7 Python abstraction for attack primitives

For now we mostly developed command-line tools that can be combined through pipes in a shell, and this solution has some limitations because we cannot build complex processing chains using only the command line. It would be very convenient to allow users to implement such complex processing chains with a Python script similar to how *GNURadio* works.

This feature has not yet been implemented in WHAD but is in the development roadmap, and interoperability with GNURadio is actually another possibility in discussion. Both GNURadio and WHAD make use of modular blocks/primitives that can be combined to build complex processing chains.

3.8 Evaluation and perspectives

The implementation of the different attack primitives in *WHAD* simplifies the creation of tools dedicated to a specific protocol and facilitates the exploration of unknown wireless protocols. The modularity of the framework combined with the use of a well-known scripting language like Python, the fact that it relies on *Scapy* for packet dissection and crafting and the reuse of existing code originally implemented in *Mirage* makes it a powerful and flexible toolbox. The limited set of tools derived from primitives of the proposed systematization and the way they can be combined to build complex attack scenarios encourages the development of compatible tools and participates in reducing the fragmentation.

However, this approach and the proposed implementation (presented at DEFCON 32 [30]) suffer some limitations:

— Python is a powerful scripting language but quite slow at execution time, which combined with the architecture of *WHAD* introduces a non-negligeable latency that is incompatible with time-constrained operations required by some wireless protocols.

— *WHAD*'s piping mechanism also adds some overhead and slightly impacts the framework efficiency, while providing a lot of flexibility.

Other security researchers have been using *WHAD* for their current research works:

— Orlaine Guetsa, Alexandre Goncalves and Morgan Yaklehef presented at *leHACK* in 2023 how they discovered and exploited multiple vulnerabilities in a Bluetooth Low Energy padlock [27].

— Baptiste Boyer fuzzed different Bluetooth Low Energy GATT implementation with *WHAD* and presented its work at *Hardwear.io NL 2024* [4].

— Pierre Ayoub experimented with *Screaming Channel* attacks on a real-world firmware of a BLE-enabled device to determine the conditions required for such attacks to succeed [28].

— Elies Tali presented his work on the *Bluetooth Mesh* protocol and its vulnerabilities at SSTIC 2025 [12], including tools and exploits based on *WHAD*.

To overcome the observed latency issues, we improved the WHAD protocol to include a new feature to manage transmission of prepared *PDUs* when specific events occur, directly in the firmware. Pierre Ayoub used this specific feature to optimize his attacks because of the required precise timing.

A Rust-based implementation of *WHAD* has also been started in 2025, in an attempt to fix the performance issues observed with the Python-based implementation, but is still in early stage of development.

4 Conclusion

This paper introduces a systematization of attacks against wireless networks based on an analysis of common known attacks. This analysis identified eleven attack primitives used on two main layers: the link layer where connections and peer-to-peer communications are defined, and the physical layer at the interface of the communication medium and the link layer. The *Capture* and *Injection* primitives are defined at the physical layer whereas the *Connection*, *Synchronization*, *Spoofing* and *Jamming*

primitives are associated with the link layer. Five more primitives named *Dump*, *Replay*, *Forge*, *Extraction* and *Transform* are transversal and can be used on both layers. Attacks against wireless networks have been described using these primitives, using specific combinations and sometimes time-dependent actions.

These primitives can also describe basic tools that can be used to perform the modeled attacks, and we put them to the test while we were developing *WHAD*, an open-source framework aiming at playing with wireless protocols and related attacks. We demonstrated that, with carefully designed tools, it is possible to implement multiple attacks without creating complex tools but rather by leveraging a limited set of tools to implement these attack primitives. This approach also empowers users with greater flexibility and creativity, enabling them to design and implement new attacks by building tools upon these primitives.

References

1. Aircrack-ng. https://www.aircrack-ng.org/.
2. Sarmed Almjamai. A Comprehensive Taxonomy of Attacks and Mitigations in IoT Wi-Fi Networks. https://www.diva-portal.org/smash/get/diva2:1719628/FULLTEXT02, 2022. [Online; accessed 12-Dec-2024].
3. Daniele Antonioli. Bluffs: Bluetooth forward and future secrecy attacks and defenses. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 636–650, New York, NY, USA, 2023. Association for Computing Machinery. https://doi.org/10.1145/3576915.3623066.
4. Baptiste Boyer. Bluetooth low energy gatt fuzzing: from specification to implementation. https://hardwear.io/netherlands-2024/speakers/baptiste-boyer.php, 2024.
5. Damien Cauquil. Btlejack. https://github.com/virtualabs/btlejack.
6. Damien Cauquil. Btlejuice. https://github.com/DigitalSecurity/btlejuice.
7. Romain Cayre. Mirage. https://github.com/RCayre/mirage.
8. Romain Cayre. InjectaBLE: Injecting malicious traffic into established Bluetooth Low Energy connections. https://laas.hal.science/hal-03193297v2, 2021. [Online; accessed 12-Dec-2024].
9. Romain Cayre. Offensive and defensive approaches for wireless communication protocols security in IoT. https://laas.hal.science/tel-03841305v2/file/2022RomainCAYRE.pdf, 2022. [Online; accessed 17-Dec-2024].
10. Romain Cayre, Vincent Nicomette, Guillaume Auriol, Eric Alata, Mohamed Kaâniche, and Geraldine Marconato. Mirage: towards a Metasploit-like framework for IoT. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, Berlin, Germany, October 2019. https://laas.hal.science/hal-02346074.

11. Dominic Spill. Ubertooth. https://github.com/greatscottgadgets/ubertooth.

12. Romain Cayre Vincent Nicomette Elies Tali, Guillaume Auriol. Tous les chemins mènent à drop : une évaluation de la sécurité d'un mécanisme de routage du bluetooth mesh. https://www.sstic.org/2025/presentation/tous_les_chemins_mnent__drop__une_valuation_de_la_scurit_dun_mcanisme_de_routage_du_bluetooth_mesh/, 2025.

13. Gowri Sankar Ramachandran Stéphane Delbruel Wouter Joosen Danny Hughes Emekcan Aras, Nicolas Small. Selective jamming of lorawan using commodity hardware. https://hal.science/hal-04901601v1/document, 2017.

14. GNU Radio. GNU Radio. https://www.gnuradio.org/.

15. Vinay M. Igure and Ronald D. Williams. Taxonomies of attacks and vulnerabilities in computer systems. *IEEE Communications Surveys & Tutorials*, 10(1):6–19, 2008.

16. Stephen Thomas Kent. Encryption-based protection for interactive user/computer communication. In *Proceedings of the Fifth Symposium on Data Communications*, SIGCOMM '77, page 5.7–5.13, New York, NY, USA, 1977. Association for Computing Machinery. https://doi.org/10.1145/800103.803345.

17. SEEMOO Lab. InternalBlue, a Bluetooth experimentation framework for Broadcom and Cypress chips. . https://github.com/seemoo-lab/internalblue.

18. Karim Lounis and Mohammad Zulkernine. Attacks and defenses in short-range wireless technologies for iot. *IEEE Access*, 8:88892–88932, 2020.

19. Satya Prakash Yadav Manish Kumar, Vibhash Yadav. Advance comprehensive analysis for Zigbee network-based IoT system security. https://link.springer.com/content/pdf/10.1007/s10791-024-09456-3.pdf, 2024. [Online; accessed 12-Dec-2024].

20. Marc Newlin. MouseJack: Injecting Keystrokes into Wireless Mice. https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEF%20CON%2024%20-%20Marc-Newlin-MouseJack-Injecting-Keystrokes-Into-Wireless-Mice.pdf.

21. John McHugh. The 1998 lincoln laboratory ids evaluation. In Hervé Debar, Ludovic Mé, and S. Felix Wu, editors, *Recent Advances in Intrusion Detection*, pages 145–161, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

22. Roy McNamara. Networks — where does the real threat lie? *Information security technical report.*, 3(4):65–74, 1998.

23. Marcus Mengs. LOGITacker. https://github.com/RoganDawes/LOGITacker.

24. Meshtastic. Meshtastic, an open source, off-grid, decentralized, mesh network built to run on affordable, low-power devices. https://meshtastic.org/.

25. Meshtastic. Meshtastic frequency slot calculator. https://meshtastic.org/docs/overview/radio-settings/#frequency-slot-calculator.

26. Meshtastic. Meshtastic github repository. https://github.com/meshtastic/firmware.

27. Morgan Yaklehef Orlaine Guetsa, Alexandre Goncalves. Vulnerability analysis of a bluetooth low energy padlock. https://lehack.org/2023/track/vulnerability-analysis-of-a-bluetooth-low-energy-padlock/, 2023.

28. Aurélien Francillon Clémentine Maurice Pierre Ayoub, Romain Cayre. Bluescream: Screaming channels on bluetooth low energy. https://hal.science/hal-04725668v1, 2024.
29. RiverLoopSec. Killerbee, a IEEE 802.15.4/ZigBee Security Research Toolkit. https://github.com/riverloopsec/killerbee.
30. Damien Cauquil Romain Cayre. One for all and all for WHAD: wireless shenanigans made easy. https://media.defcon.org/DEF%20CON%2032/DEF%20CON%2032%20presentations/DEF%20CON%2032%20-%20Damien%20Cauquil%20Romain%20Cayre%20-%20One%20for%20all%20and%20all%20for%20WHAD%20wireless%20shenanigans%20made%20easy.pdf.
31. Mike Ryan. Crackle. https://github.com/mikeryan/crackle.
32. Scapy. Scapy. https://scapy.net/.
33. Securing. Gattacker, A Node.js package for BLE (Bluetooth Low Energy) security assessment using Man-in-the-Middle and other attacks . https://github.com/securing/gattacker.
34. William Stallings. *Network and internetwork security: principles and practice*. Prentice-Hall, Inc., USA, 1995.
35. Mohsan Azeem Muhammad Farhan Sana Naseem Bushra Mohsin Tahira Ali, Rashid Baloch. A Systematic Review of Bluetooth Security Threats, Attacks & Analysis. https://www.ijcttjournal.org/2021/Volume-69%20Issue-7/IJCTT-V69I7P101.pdf, 2020. [Online; accessed 12-Dec-2024].
36. Thorsten Schroeder, Max Moser. Practical Exploitation of Modern Wireless Devices. http://www.remote-exploit.org/articles/keykeriki_v2_0__8211_2_4ghz/, 2010.
37. Mark Vink. A Comprehensive Taxonomy of Wi-Fi Attacks. https://www.cs.ru.nl/masters-theses/2020/M_Vink___A_comprehensive_taxonomy_of_wifi_attacks.pdf, 2020. [Online; accessed 12-Dec-2024].
38. Wikipedia. Kiss principle. https://en.wikipedia.org/wiki/KISS_principle.
39. Wikipedia. OSI model. https://en.wikipedia.org/wiki/OSI_model. [Online; accessed 12-Dec-2024].
40. Wikipedia. Wired Equivalent Privacy. https://en.wikipedia.org/wiki/Wired_Equivalent_Privacy. [Online; accessed 17-Dec-2024].