INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Streaming Multicast Authentication with TESLA and ESP on Linux

Jonas Dellinger

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Streaming Multicast Authentication with TESLA and ESP on Linux

Jonas Dellinger

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Dr. Nils gentschen Felde Tobias Guggemos
Abgabetermin:	12. November 2018

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 12. November 2018

. . . .
(Unterschrift des Kandidaten)

Abstract

The Internet of Things (IoT) connects devices of different sizes, including very small-scale and constrained devices. While certain resource-intensive tasks can be outsourced to more powerful devices, establishing and maintaining an authenticated communication will always be a requirement for nodes of the IoT ecosystem. Achieving multicast origin authentication is a particular and perpetual challenge in the present time. A proposed solution is the usage of the Timed Efficient Stream Loss-Tolerant Authentication (TESLA) protocol. It is based exclusively on symmetric cryptography and provides multicast origin authentication to receivers in a delayed manner. Within the scope of this thesis, TESLA was implemented and integrated with the IPsec transport protocol Encapsulating Security Payload (ESP) on a Linux based platform. In addition, possible bootstrap processes were designed with the help of existing IPsec protocols, including IKEv2 and G-IKEv2. It was tested and evaluated based on different scenarios with the help of the FIT IoT-LAB. As a result, two prototype TESLA and ESP libraries have emerged and can be used in future projects.

AEAD	Authenticated Encryption with Associated Data
AH	Authentication Header
DoS	Denial of Service
EMSS	Efficient Multi-chained Stream Signature
ESP	Encapsulating Security Payload
HMAC	Keyed-Hash Message Authentication Codes
IANA	Internet Assigned Numbers Authority
IBS	Identity-Based Signatures
ICV	Integrity Check Value
IETF	Internet Engineering Task Force
IKE	Internet Key Exchange Protocol
IKEv2	Internet Key Exchange Protocol V2
G-IKEv2	Group Internet Key Exchange Protocol V2
IKE	Internet Key Exchange
IoT	Internet of Things
IPsec	Security Architecture for the Internet Protocol
IV	Initialization Vector
MAC	Message Authentication Code
μTESLA	μTESLA
PKCS#5	Password-based Encryption Standard
SA	Security Association
SAD	Security Association Database
SNEP	Secure Network Encryption Protocol
SPD	Security Policy Database
SPI	Security Parameters Index
TESLA	Timed Efficient Stream Loss-Tolerant Authentication
TS	Traffic Selector
TTP	Trusted Third Party
SHA	Secure Hash Algorithm

NIST	National Institute of Standards and Technology
PRF	pseudo-random function
FOSS	Free Open-Source Software
API	Application Programming Interface
PSK	pre-shared key
RTT	Round Trip Time
ECDSA	Elliptic Curve Digital Signature Algorithm
GCKS	Group Controller / Key Server
GSA	Group Security Association
KD	Key Download
TEK	Data Security Key
DH	Diffie–Hellman key exchange

Contents

1 Introduction

Embedded Systems have an even bigger impact in our lives than ever before. While mobile phones are getting more powerful every year, the Internet of Things (IoT) has different demands for its systems. They should be energy-efficient, small-scale and able to communicate with other similar devices. Examples for such devices are smart home systems, which include wireless light bulbs and heating, or smart assemble factories consisting of multiple independently operating devices. To fulfill the above mentioned demands, devices are utilized, constrained in terms of memory size and compute power. Although it is possible to offload heavy calculations to more powerful systems, the constrained device still has to perform tasks, like establishing a secure communication channel to other devices, using their own resources.
Regarding the example of establishing a secure connection, one of the key components is authentication. It makes sure a receiver can confirm that the data was sent by the correct, trusted sender. There are several protocols ready to be used, but most of them are based on complex asymmetric cryptographic functions and require unicast usage. In a multicast environment, source authentication is even more important than in a unicast one, simply because a single malicious packet can reach millions of devices instead of just one.

In traditional unicast authentication protocols, like IPSec and TLS, hybrid cryptosystems are used to speed up the exchange of data [DR08] [KS05]. They utilize symmetric cryptography for fast authentication/encryption of the actual data and asymmetric cryptography for the initial handshake and secret key exchange. When trying to apply a simplified version of this kind of authentication to a multicast scenario, we clearly run into a problem: A single packet from the server could reach multiple receivers, so every receiver needs to know the secret key used to authenticate the data. As soon as that is the case, a receiver can easily impersonate the sender and forge messages.

Another common way to achieve authentication is by using just digital signatures, which for example are provided by the public-key cryptosystem RSA [RSA78]. The sender uses its private key to generate the signature of the message. Every receiver then can verify the message by verifying the signature against the senders public key. This would also work in a broadcast scenario, but digital signatures have two notable drawbacks for the IoT World. First of, they need a lot of computational overhead for signing and verifying. Furthermore, the resulting signature adds a lot of overhead to the network. For example, a signature generated with RSA2048 would be 256 bytes long, even if the message itself is just 2 bytes. More modern algorithms, like Elliptic Curve Digital Signature Algorithm (ECDSA), are able to produce shorter and equally secure signatures, but often implicate worse sign and verify speeds [Cry]. Compared to a faster and similar secure symmetric Message Authentication Code (MAC), which can be as short as 32 bytes, this is not feasible on a lot of smaller scale embedded systems.

1.1 Timed Efficient Streaming Loss-Tolerant Authentication

In June 2005, Adrian Perrig, Ran Canetti, Dawn Song, J. D. Tygar and Bob Briscoe proposed a new authentication protocol called TESLA. Like the name suggests, it provides delayed source-authentication and integrity-checking of each packet received by a multicast/broadcast stream. To operate, it requires every receiver to be loosely time-synchronized and bootstrapped by the sender once. However, the sender does not have to keep a per-client state which makes it scaleable in large networks. Additionally, all operations in TESLA, including signing and verifying, solely use symmetric cryptography, which makes it suitable for small scale hardware. The basic authentication scheme follows the principle of delayed disclosure: The sender calculates a MAC over a payload, attaches it and sends it. The secret key used is not yet known to the receiver. Thus, it has to buffer all incoming packets first. In a later packet, the secret key is disclosed and receivers are able to authenticate buffered packets. This procedure is strictly tied to the underlying time synchronization, action as the asymmetric factor. Nonetheless, TESLA is no silver bullet. It requires an external authenticated bootstrap before its usage and at least for the sender, its memory size requirement is rather high and maybe can not be fulfilled by very small IoT devices.

In the IoT and sensor network world, TESLA could be an interesting and considerable choice as soon as a multicast streaming authentication protocol is needed. In the case of a more powerful base station and multiple smaller scale receivers, TESLA's full potential can be utilized without drawbacks. Setups like this can be found frequently in the above mentioned scenarios. For example, it can be found in the smart lighting system Phillips Hue, where a Raspberry Pi sized bridge controller is used to communicate with smaller light bulbs [Hol18]. While no further system specifications are publicly available, the rather small scale of such light bulbs could indicate the impracticality of asymmetric cryptography. For example, by creating a group of light bulbs for every room, the system could utilize TESLA to efficiently and securely send instructions to each member of the group via an IP multicast network. To further require a continuous stream of data in such a setup, one can add the functionality of multiple quick light changes in a short time to the feature list. This allows the lightning system to be used for music- or TV-based light visualizations.
Currently, however, there is no public TESLA library or application implementation available.

1.2 Outline

As part of this thesis, the integration of the TESLA protocol into the transport protocol ESP is designed, implemented and evaluated. The integration and implementation is based on the Linux operating system. Raspberry Pis Version 2 and ARM based A8 nodes running on the FIT IoT-LAB[IL18b] act as the underlying testing platform. While these devices are more powerful than standard small scale embedded IoT devices, they can be seen as high performance control nodes.

The thesis starts with a small introduction of hash functions, asymmetric and symmetric authentication and the Security Architecture for the Internet Protocol (IPsec) suite. This will setup fundamental knowledge for the underlying techniques used by TESLA and ESP. Following up, the TESLA protocol will be explained in detail, including it's authentication operations, general properties and requirements. Additionally, a list of related multicast authentication protocols is presented to the reader. Based on this knowledge, the next chapter serves as an introduction into the design of the upcoming implementation, covering all roles, components and the overall communication between them, followed by a presentation of the actual implementation, including the explanation of important code parts, testing against other implementations and relevant analysis. The thesis closes with proposals for additional and future work on the implementation and topics related to multicast authentication.

2 Background

To establish a solid background knowledge required for the thesis, this chapter will first introduce into basic cryptographic hash operations. With this knowhow, an overview of a hash based data structure, called hash chains, is provided. It continues with an explanation of how basic symmetric and asymmetric authentication can work in unicast and multicast networks. The chapter ends with a detailed look at the protocol suite IPsec, which includes but is not limited to the IP transport protocol ESP.

2.1 Hash Functions

A hash function is used to compute a fixed size hash/digest from a message of arbitrary length. It can be as simple as a modulo operation. Considering $x \mod 10$, one can input any positive number and will receive a number in the range from 0 to 9. Complex hash functions on the contrary use a combination of multiple operations, like AND, XOR, ROT, ADD and NOT. Hashes are used in various domains of computer science. They enable fast database lookups and collision detection. Furthermore, hash functions are also important elements of multiple cryptographic algorithms, for example when generating MACs or digital signatures. However, not every hash function can be used in cryptography. Cryptographic hash functions have to satisfy additional properties where the most common are the following [TK05]:

- **Collision Resistance**: Finding two messages which yield the same digest should be infeasible

- **Preimage Resistance**: Knowing a digest, it should be infeasible to compute a message yielding the same digest.

- **Second-Preimage Resistance**: Knowing a message and its hash, it should be infeasible to compute a different message yielding the same hash.

- **Fast**: cryptographic hash functions are often used in symmetric authentication, they should be able to calculate a digest rather fast.

Referring to our $x \mod 10$ hash function example, it becomes clear that it is not suitable as a cryptographic hash function.

Secure Hash Algorithm (SHA) is an example for a family of cryptographic hash functions, published by the National Institute of Standards and Technology (NIST). It contains commonly known secure hash algorithms, like SHA2 and its latest member SHA3. By now, some of the older functions, like MD5 and SHA1, are considered insecure, since collisions have been found [SBK$^+$17][Ste06].

2.2 Hash Chains

A cryptographic hash function can also be used to construct a chain of hashes. To set up a hash chain K of length n, a seed s and a hash function F needs to be chosen. s will be the first input to F and its resulting digest will be recursively put into the hash function n times.

$$\begin{aligned} K_0 &= s \\ K_1 &= F(s) \\ K_2 &= F(K_1) \\ &... \\ K_n &= F(K_{n-1}) \end{aligned} \tag{2.1}$$

Due to the one-way (injective) property of F, it is infeasible to guess or calculate K_{i-1} knowing K_i. However, it is easy to check whether K_{i-1} is a member of the chain knowing K_i by applying F on it and comparing the result. This property makes it also suitable for one-time password systems, which can be used to authenticate users. An example for such a system is shown by [Lam81]:

1. The seed s is randomly chosen by the user and a hash chain K of length n is generated based on it. n determines how many authentications are possible. A counter $i = n$ is required as well.

2. The last element of the chain, K_n is sent securely to the server and is persistently stored.

3. Once the user wants to authenticate himself, he sends K_{i-1} to the server, which can check whether its a valid hash chain element by applying F and comparing the result with the stored value. After successful authentication, the server stores the new value K_{i-1} instead of K_i

4. The client decrements its counter $i = i - 1$ and continues with step *3*, once he wants to authenticate again.

2.3 Authentication

One of the key requirements in every secure network is to make sure that the message received originally has been sent by a specific trusted sender. This act of confirming the truth is called origin authentication. There are many levels of authentication, for example single-factor and two-factor authentication. These levels specify the amount of knowledge needed to completely confirm the process of authentication. For example, a single-factor level requires a single shared secret key or password, whereas a two-factor level requires an additional property, e.g. a One-Time password [MMPR11]. When trying to secure a communication across a network, commonly a single-factor level with either symmetric or asymmetric cryptography is used. Both types of algorithms ensure integrity and authentication.
When using symmetric based authentication, sender and receiver need to have access and agree to the same shared key k. With this key k, the sender is able to generate a Message Authentication Code (MAC) *mac* over the message *msg* using a previously chosen algorithm:

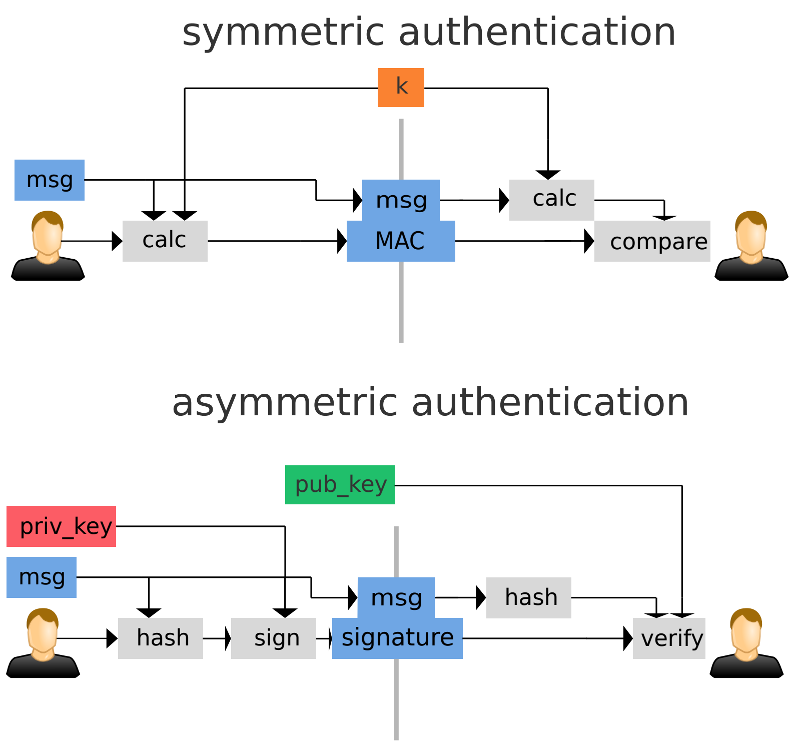


Figure 2.1: Visualization of asymmetric and symmetric authentication

e.g. *Keyed-Hashing for Message Authentication* (HMAC)[KBC97] in combination with the hashing function *Secure Hash Algorithm 256* (SHA-256). Concatenating *msg* and *mac* will result in a *packet* which can be sent to the client. Upon receiving the packet, the client deconstructs *msg* and *mac* and checks whether his calculation using k and *msg* yields the same *mac* as the one received. This method in comparison to asymmetric authentication allows fast verifying and signing but requires a pre-shared key.
On the contrary, asymmetric authentication requires each sender of the communication to have a public and private key (respectively pub_{key} and $priv_{key}$). These can be generated by algorithms like RSA[JK03]. Since asymmetrically signing large data is expensive, the sender first generates a hash over the original message *msg*. This hash then gets signed with the senders private key $priv_{key}$, resulting in a *signature*. Both, *msg* and *signature* will be concatenated and sent to the receiver, who is then able to verify the message by first hashing the *msg* and then using this hash and the senders public key pub_{key} to completely validate the message. Whereas asymmetric signing and verifying processes are more expensive than symmetric ones, they provide an additional safety-property called non-repudiation as long as the secret keys are not exposed. This is not the case for symmetric authentication since the other participant is also able to generate authentic messages using the shared secret key k.
Figure 2.1 shows a side-by-side visualization of both methods. To eliminate each protocols' disadvantages and combine their advantages, authentication and encryption protocols like TLS use a hybrid approach[DR08]. The initial session setup and symmetric key exchange will be done with asymmetric cryptography and further session communication will be exchanged with a symmetric algorithm. This also allows to incorporate a certificate authority,

which will act as a Trusted Third Party (TTP) and permits participants to validate the ownership of public keys.

2.4 Multicast Authentication

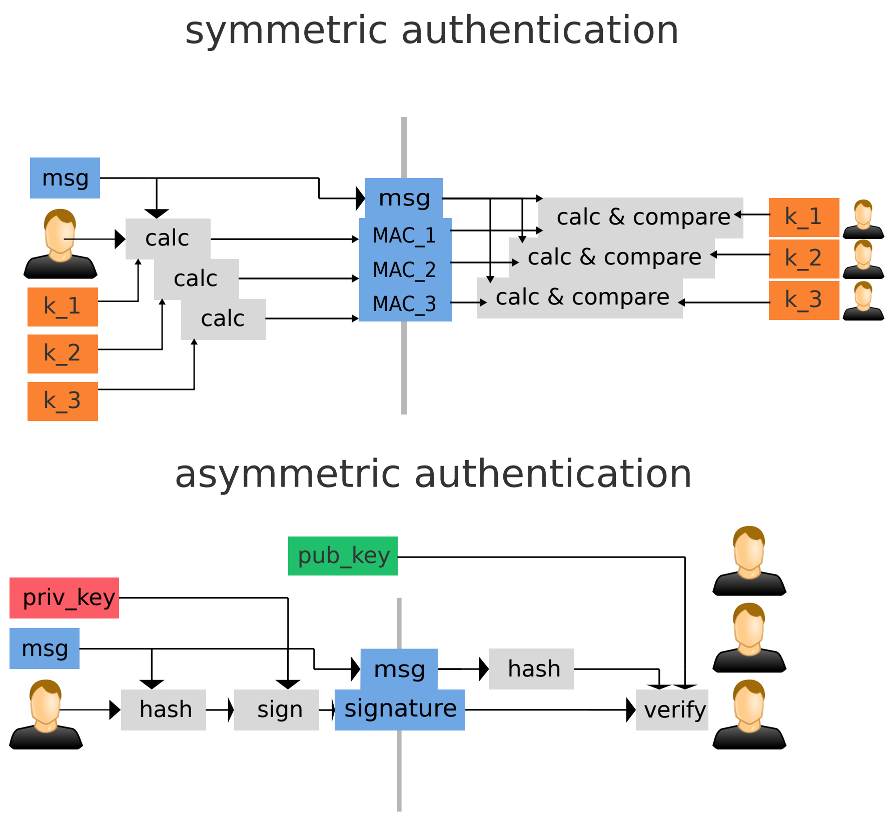


Figure 2.2: Visualization of asymmetric and symmetric multicast authentication

The last chapter introduced into basic unicast authentication using asymmetric and symmetric cryptography. Using the same cryptographic operations, the basic scheme can also be applied to multicast communication, which allows multiple receivers to take part:

Symmetric based authentication works in a similar manner. However, a single secret key shared between receivers and sender is not sufficient any more. Any receiver could make use of the shared key to send authentic messages to other receivers. Instead, the sender must share a unique shared key with every receiver and furthermore, a message by the sender now has to contain a MAC for every receiver. This required change introduces a linear dependency between overall size per message, shared keys and the number of receivers. Thus, this approach does not scale very well for a large amount of receivers.

Asymmetric authentication, on the contrary, does neither require any message size changes, nor more public or private keys. Receivers simply need to have access to the sender's public key. Since there is no dependency between number of receivers and number of keys or message size, this approach scales very well for a large amount of receivers.

Similar to figure 2.1, figure 2.2 shows the same process, but in case of an authenticated multicast broadcast with 3 receivers.

2.5 Internet Protocol Security

When considering a secure network message exchange, authentication is just one part of it. A primarily key exchange, encryption and replay protection are often required as well. IPsec is an open standard developed by the Internet Engineering Task Force (IETF)[KS05]. It defines two primary traffic security protocols, Authentication Header (AH) and ESP, for secure layer-3 IP based network communication. Both protocols utilize the security services and secret keys provided by a Security Association (SA), which in turn is stored in a Security Association Database (SAD).

2.5.1 Transport Modes

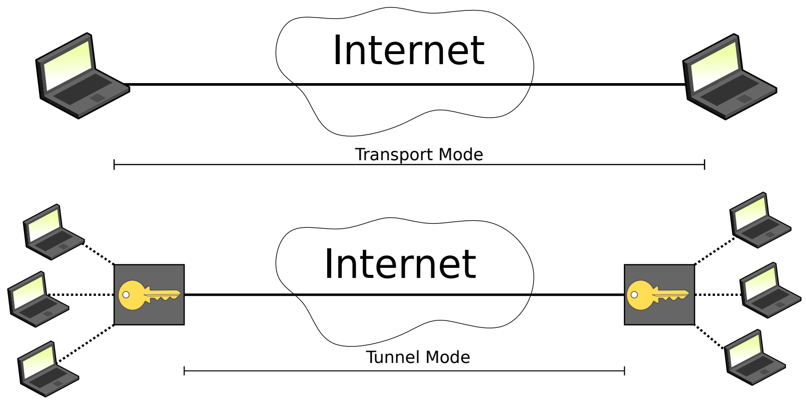


Figure 2.3: Transport Mode and Tunnel Mode

AH and ESP can be operated in two different modes: Tunnel Mode and Transport Mode. Which mode is used for a connection depends on the entry in the corresponding SA.

When operating in Tunnel Mode, no changes to the original IP packet are made. Instead, the original packet is embedded in a newly created protocol header, AH or ESP, which then in turn gets embedded in an IP packet. This packet has the next header value of *50* for ESP and *51* for AH. Because the original packet can be restored at the destination, this mode is suitable for Gateway-to-Gateway communication (Figure 2.3).

In Transport Mode, the original IP Header is reused with a single modification to the *Next Header* field. It is changed according to the used protocol. The original IP payload is then exchanged by the chosen protocol header which includes the original IP payload respectively. This mode is typically used in end-to-end communication (Figure 2.3)

2.5.2 Encapsulating Security Payload

IPsec provides ESP, an IPv4 and IPv6 traffic protocol, which is defined in [Ken05b]. It provides various security services:

- Origin authentication
- Confidentiality
- Stateless integrity

- Replay protection
- Traffic flow confidentiality

However, the actual used security services depend on the underlying SA. It determines how the ESP header will be generated/processed for incoming/outgoing packets. With a single restriction, explained in 2.5.3, one would be able to use the header without any security service. Figure 2.4 shows a detailed view of the header with all of its fields, which will be explained below.

Security Parameters Index - Since the header does not contain any information regarding used security services, this index is required to make sure the receiver knows how to process an incoming header. The triplet, consisting of the Security Parameters Index (SPI), the destination IP address and the traffic protocol ESP, result in a unique index and can be used to identify the corresponding SA. In this way, the receiver is able to look up the algorithms and secret keys which were involved when the header was generated. This allows for further processing, like performing authentication or decryption.

Sequence Number - To achieve replay protection, ESP uses a sequence number. Once the corresponding SA is negotiated, both sides have to initialize it. During this initialization, the *sequence number* field has to be set to 0. Depending on whether the SA was configured to support replay protection, the sender needs to make sure that he increases the *sequence number* field every time he sends a packet and checks that he does not use the same number twice. On behalf of the receiver, one needs to check whether the sequence number fits into a sliding window and has not been received already. If replay protection is disabled, the receiver should ignore the field. Once the sequence number reaches its maximum, a new SA needs to be negotiated. This incidentally also makes sure that authentication and encryption keys are replaced periodically.

Payload Data - This field contains the payload data which is secured by ESP. For example, it could be a TCP or UDP Header. The header has no dedicated Initialization Vector (IV) field but supports encryption algorithms which depend on one, which is the reason why the first x bytes of the payload data are often interpreted as the current IV. This depends on the configuration of the underlying SA. It is first field in the header, which is covered by optional confidentiality.

Padding - The *next header* field has to be aligned correctly, not depending on the size of the payload data. Therefore, a special padding field is needed. This field is not always required and can have a length of zero, for example if the payload size is divisible by 6 bytes. Furthermore, this field is also necessary for various encryption algorithms, which need the payload size to be a multiple of some size (This is the case for block cipher algorithms). Most encryption algorithm implementations already support padding, but not in a standardized way. OpenSSL uses Password-based Encryption Standard (PKCS#5) padding, which will assign each padded byte the value of the overall padding length [Opea]. To bypass implementation differences and to get correctly aligned fields, ESP specifies that padding needs to be implemented utilizing the *padding* and *padding length* field. This can be achieved by correctly applying ESP padding before feeding the payload into the encryption algorithm implementation.

Padding Length - To differentiate between actual payload and padding data, the *padding length* field is used. It holds a 2 byte value specifying how many padding bytes have been added.

Next Header - After processing the packet, the receiver needs to know which type of payload he received and how to further process it. The *next header* field stores the number value of the next header, which is standardized by the Internet Assigned Numbers Authority (IANA) and published in [Int]. In the case of TCP or UDP, the number would be 6 or 17.

Integrity Check Value - ESP has to support stateless integrity and origin authentication. This is achieved with a *Integrity Check Value (ICV)* field, which often contains a MAC or Keyed-Hash Message Authentication Codes (HMAC), depending on the SA configuration. When authentication is disabled, this field must be empty and will be ignored by the receiver.

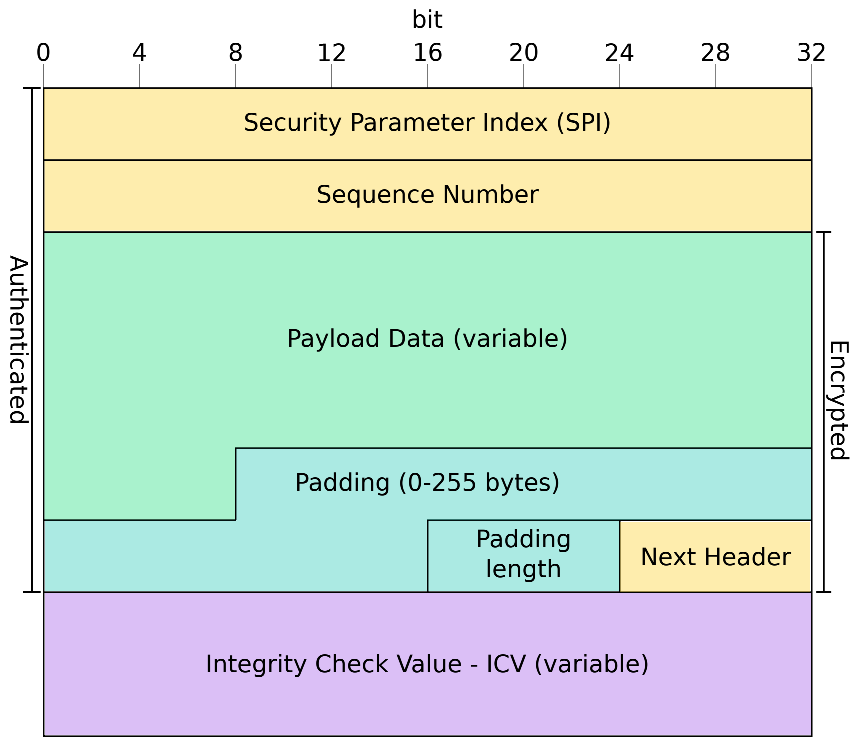


Figure 2.4: ESP Header

2.5.3 Encapsulating Security Payload Algorithms

ESP specifies which algorithms need to be supported by an implementation and if they can be used in combination [WMM$^+$17]. As mentioned above, this opens up the possibility to use only a subset of security services, depending on the use case. A SA which specifies the use of ESP with *HMAC-SHA1-96* authentication and *NULL* encryption would yield in authenticated packets without any encryption. However, the standard also forbids certain combinations, for example using ESP with *NULL* encryption and *NULL* authentication, which would yield no security service at all. [WMM$^+$17] also specifies and encourages the usage of combined mode algorithms, which support Authenticated Encryption with Associated Data (AEAD). Instead of separating encryption and authentication, a single algorithm is used to provide both security services.

2.5.4 Authentication Header

AH is specified by [Ken05a] and is a traffic protocol which provides data integrity, origin authentication and optional replay protection by utilizing a sliding window. Since it does not provide any confidentiality, it is inferior to ESP regarding its overall security services. While its header structure is similar to the ESP one, in transport mode it is also providing authentication for some immutable fields of the surrounding IP header, which ESP is not capable of.

2.5.5 Security Association and operating IPsec

To create and process ESP or AH headers, IPsec uses SAs to describe which algorithms and which secret keys need to be used. A SA is always mapped to one unidirectional connection, which implies that as soon as ESP or AH is used, a bidirectional communication requires a minimum of two SAs per host. The following notable properties are contained inside a SA:

SPI - An 32 bit sequence which loosely identifies a SA. It is considered loosely because on single hosts, they are able to uniquely identify a SA, but not on (NAT-)Gateways: "In particular, simply storing the (remote tunnel header IP address, remote SPI) pair in the SPD cache is not sufficient since the pair does not always uniquely identify a single SAD entry. For instance, two hosts behind the same NAT could choose the same SPI value."[KS05]

Sequence Number Counter - A counter used for generating outgoing sequence numbers.

AH - A list of properties regarding AH, for example if it is enabled, which authentication algorithm is used and which secret key.

ESP - Contains values similar to the AH properties, but also includes possible encryption and AEAD properties.

Lifetime - A field specifying how long the SA can be used. It also contains information on what happens once it expires, like negotiating a new SA or destroying it completely.

Mode - Whether Transport or Tunnel Mode is used when sending or receiving packets.

IP Source and Destination - When Tunnel Mode is selected, both IP addresses must be supplied, so that a new IP header can be constructed when tunneling IPsec packets. This field is not used in Transport Mode.

However, the SA for Internet Key Exchange Protocol V2 (IKEv2) is an exception: It stores configuration for a bidirectional communication and therefore contains, for example, two SPIs. Since a host should be capable of storing multiple SAs, they are stored in a special database, called Security Association Database. Generally, there are two different ways to add a new SA to it: First, using out of band negotiated protocols and secret keys, two hosts can establish a secure channel via IPsec by adding four SAs in total, manually. However, this has the disadvantage of not being able to automatically refresh secret keys once the SAs lifetime is exceeded. Secondly, Internet Key Exchange Protocol (IKE), a key management protocol, can be used to negotiate protocols and secret keys (SA) between two hosts. It also supports the usage of pre-shared keys for the negotiation. This allows an automatic exchange of new keys once the lifetime of the negotiated SA expires.

To fully operate IPsec, another database, the Security Policy Database (SPD), is required. Each of its entries is specifying a policy, containing at least one Traffic Selector (TS), a link to a SAD entry, an *on-match* action, a corresponding traffic protocol and additional optional parameters like its lifetime. Depending on the TS, a policy can match an outgoing or incoming IP packet. The configuration can consist of source and destination addresses, the next header protocol number or even fields of the next header. Every time a packet is processed by the IPsec stack, the policy of the first TS which matches will be used as further processing information. Its *on-match* action will be performed, which can be one of the following:

- **DISCARD**: Discard the packet and abort further processing.
- **PROTECT**: Use the linked SA to protect the packet with certain security services
- **BYPASS**: Bypass IPsec security but continue processing the packet

Figure 2.5 gives a rough overview regarding the processing of incoming and outgoing packets.

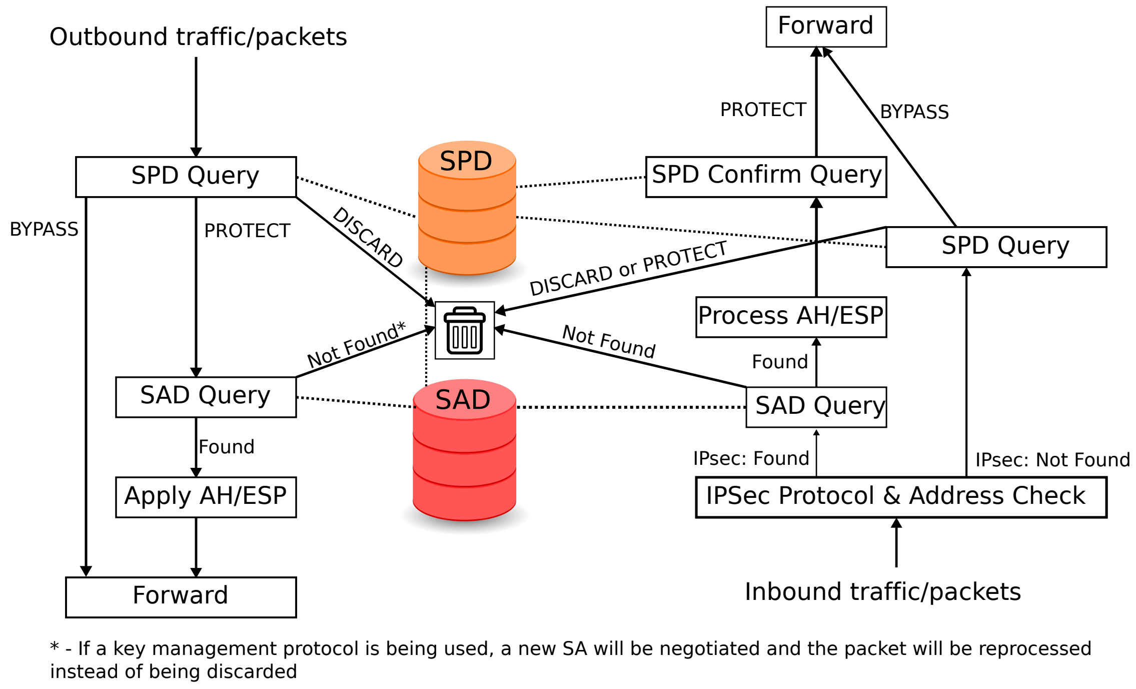


Figure 2.5: IPsec Inbound and Outbound Traffic Processing Overview

2.5.6 Internet Key Exchange Protocol V2

Instead of manually securing the communication between two hosts by statically configured SAs, a key exchange protocol like IKE can be used. Its latest version is specified by [KHN$^+$14] and is referred to as IKEv2. Its key tasks are the secure negotiation, establishing and rekeying of ESP or AH SAs (child SAs) between a client and a server. This is achieved with 3 major exchange types, which always consist of a request and a response message:

- **IKE_SA_INIT**: The client sends this message, showing interest in a new IKEv2 exchange. Together with the response of the server, both parties negotiated cryptographic algorithms, exchanged nonces and made a Diffie–Hellman key exchange (DH) [KHN$^+$14].

- **IKE_AUTH**: By sending the *IKE_AUTH* message, the client authenticates itself and the previously sent *IKE_SA_INIT* message. Among others, this can be achieved via digital certificates or pre-shared keys (PSKs). Similar, the host is able to authenticate itself in the response. As a result, a special IKE SA and a first optional child SA are established.
- **CREATE_CHILD_SA**: While a first child SA may already be created in the *IKE_AUTH* exchange, the exchange type *CREATE_CHILD_SA* can be used to create additional child SAs. Furthermore, this exchange is used to rekey the original IKE SA, or possible child SAs. Any required keying material for further child SAs is derived from the random nonces and DH in the first *IKE_SA_INIT* exchange.
  This message exchange is also automatically protected by the properties of the negotiated IKE SA.

2.5.7 Group Internet Key Exchange Protocol V2

Group Internet Key Exchange Protocol V2 (G-IKEv2) solves the same problem as IKEv2 does, however, not for a single secure connection between two hosts, but for a secure connection between a group of hosts [WS18]. Thus, the roles now consist of multiple group members and a Group Controller / Key Server (GCKS), which is responsible for their authentication, authorization and bootstrap of utilized security policies (ESP, AH) and its secret keys. The G-IKEv2 scheme follows the same request and response message model as IKEv2 and even uses the same initital message exchange *IKE_SA_INIT*. After that, *IKE_AUTH* is replaced with *GSA_AUTH*, still allowing the client to authenticate itself via PSKs or a digital certificate. Despite that, the response from the GCKS is quite different. It contains two important payloads:

- **Group Security Association**: This payload specifies the SAs used in the group communication (Data Security Key (TEK)). This includes the transport protocol, security algorithms and other SA related properties. However, it does not contain any secret keys. These values are also not based on the inital exchange between a group member and the GCKS, but rather on the underlying group specification. Additionally, the payload may include another SA for future group rekeying processes.
- **Key Download**: The Key Download (KD) payload contains every secret key needed for the SAs defined in the Group Security Association (GSA) payload. These keys are also based on the group specification and not on the initial exchange.

Generally, due to the well-defined and flexible header formats, the protocol allows a lot of customization. In fact, adding support for new authentication or encryption algorithm often only requires a protocol number assignment. The original specification [WS18] is still a draft and is not yet approved by the IETF, hence, there are no approved implementations yet.

3 TESLA

This chapter provides an overview over the TESLA protocol. Instead of relying on asymmetric cryptography, it provides a unique way of achieving origin authentication using primarily symmetric cryptography. However, its authentication is delayed and requires its receivers to buffer packets.

> TESLA allows all receivers to check the integrity and authenticate the source of each packet in multicast or broadcast data streams. TESLA requires no trust between receivers, uses low-cost operations per packet at both sender and receiver, can tolerate any level of loss without retransmissions, and requires no per-receiver state at the sender. TESLA can protect receivers against denial of service attacks in certain circumstances. Each receiver must be loosely time-synchronized with the source in order to verify messages, but otherwise receivers do not have to send any messages. TESLA alone cannot support non-repudiation of the data source to third parties. ([PSC+05])

First, a detailed overview of how TESLA can be configured is given. Once the sender is configured, each client needs to be bootstrapped as well. After this procedure, TESLA is ready to be used to send and verify authenticated messages. To summarize this chapter, a small protocol analysis and possible deployment scenarios are given.

3.1 Sender Configuration

Each network of (IoT) devices has different properties regarding its network speed, device resources or connectivity. Thus, TESLA provides a way to configure the core protocol to different scenarios. Before the sender can start sending authenticated messages, it has to determine and prepare the following variables:

Hash Chain K with length n: The sender needs to generate a hash chain H based on a random seed with length n and a pseudo-random function (PRF) F. Since the seed is already chosen randomly, a secure cryptographic hash function can also act as a PRF. The length of n will be one of the two factors specifying the last point of transmission before rekeying is needed and discussed in the next paragraph. For simplicity in the current and upcoming paragraphs, we will set the actual hash chain K to the reverse of H, which implies $K_0 = H_n$, $K_n = H_0$ and generally $K_i = H_{n-i}$. 3.1 shows an example of a resulting hash chain K. In contrary to the one-time password system explained in 2.2, TESLA uses its hash chain in the exact opposite: As soon as the first element K_0 is known, it's possible to check whether a value is an element of the hash chain at index i. As a concrete example, if one wants to know a received key is hash chain element 5, he can recursively apply F exactly 5 times on it and compare the result to K_0. It can be generalized in the following formula:

$$K_0 == F^i(x) \tag{3.1}$$

where $F^i(x)$ means i recursive calls to F using x as the first value. The ability to check and recalculate chain elements is also the foundation of TESLAs packet loss support which will be explained in detail in 3.5. Due to the fact, that each hash chain element will be used as a secret key when signing messages, it can also be called a key chain.

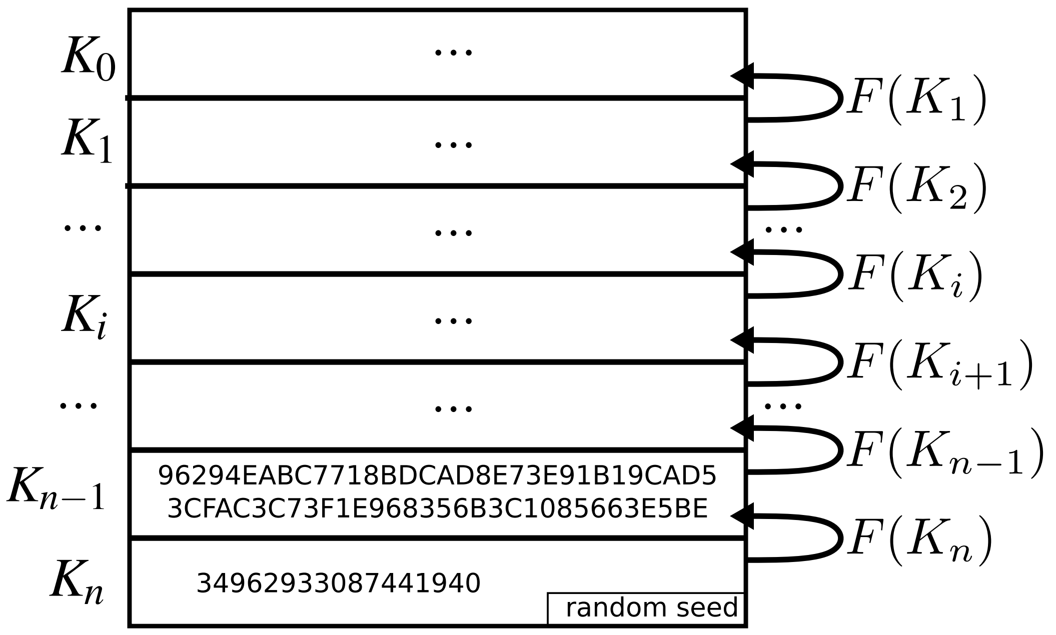


Figure 3.1: A reversed Hash Chain example K

Interval Duration T_{int}: The second factor regarding total transmission time is the interval duration time T_{int}. This value divides time into equally long time spans and specifies how long a hash chain element K_i is used. For simplicity, we will use milliseconds as its unit. The last point of transmission can be calculated by using $n \cdot T_{\text{int}}$. This is also a critical value regarding the delayed authentication: "A T_{int} that is too short will cause the keys to run out too soon. A T_{int} that is too long will cause excessive delay in authentication for some of the packets (those that were sent at the beginning of a time period)."[PSC+05] Furthermore, the original RFC [PSC+05] also provides a suggestion on how to calculate a reasonable value for T_{int}: $T_{\text{int}} = max(n, m)$ where n is the estimated delay between packets in milliseconds and m is the estimated upper bound regarding network delay in milliseconds ("[...] This (delay) includes any delay expected in the stack [...]"[PSC+05]).

Key Disclosure Delay d: The key disclosure delay d states, how many time intervals have to pass until a key chain element gets disclosed. In particular, a key chain element which was used in interval i will be disclosed in interval $i + d$. The combination of T_{int} and d specify the maximum delay a receiver has to go through when authenticating packets. For example, a T_{int} of 1 second in combination with a disclosure delay of 4 could result in an authentication delay of up to 5 seconds. The initial packet could arrive early in the interval i, and the disclosed key could arrive very late in interval $i + 4$. Since this value is critical regarding TESLAs performance as well, a formula for finding a reasonable value is given by the RFC: $d = ceil(2m/T_{\text{int}}) + 1$ where $2m$ is an upper bound for the round trip time of packets [PSC+05]. This results in a d larger than 1, which eliminates the following cases:

- $d == 0$: If d is zero, no security is provided due to instantly disclosed keys and the possibility to forge authentic messages in the same time interval.

- $d == 1$: While a d of 1 theoretically still provides security, all packets sent close to the boundary of the next interval will be dropped by receivers. In detail, upon receiving a packet, the receiver has to make sure the sender is not yet in an interval where the corresponding key has been disclosed. So a packet, which was sent close to the boundary, could arrive in the next interval, which is also the disclosure phase for that packet. The receiver would have to drop it.

With the above mentioned variables and structures set, a sender is now able to bootstrap potential receivers.

3.2 Bootstrapping Receivers

Before a receiver is able to verify authenticated packets, it needs to be bootstrapped by the server. This bootstrap includes establishing loosely time synchronization and the exchange of some TESLA variables which where defined in the previous chapter. The draft itself doesn't specify on how one achieves this bootstrap, but requires that the receivers know the following properties:[PSC+05]

Receivers need to set D_t, an upper bound regarding the clock difference between the local and the sender clock. In particular, this requires the formula $t_{sender} < t_{local} + D_t$, where t_{sender} is the current time at the sender and t_{local} is the current time at the receiver, to always hold true. Note, that D_t can be different for each receiver and needs to be calculated for each independently. The protocol does not specify how this time synchronization should be established, but it provides an exemplary method to achieve it: '

- The receiver sends a (sync t_r) message to the sender and records its local time, t_r, at the moment of sending.
- Upon receipt of the (sync t_r) message, the sender records its local time, t_s, and sends (synch, t_r,t_s) to the receiver.
- Upon receiving (synch,t_r,t_s), the receiver sets $D_t = t_s - t_r + S$, where S is an estimated bound on the clock drift throughout the duration of the session.

' ([PSC+05]). This exchange needs to happen on an authenticated communication channel, which can be achieved by using a session setup protocol.

Additionally, the following variables have to be transmitted, also in an authenticated manner:

- K_0: The first value of the used hash chain K. This value is also called commitment to the key chain K, since it allows the receiver to check whether a value is an element of the key chain K, which has been explained in 3.1. It's also possible to send K_i, where i is any index, instead of K_0. However, due to the properties of hash chains, the receiver then is only able to verify keys with an index equally or larger than i.
- T_{int}: The interval duration, which has been set by the sender in 3.1
- T_0: The time at which interval 0 will start, i.e when the transmission will start.

- n: The length of the utilized hash chain K.
- d: The specified key disclosure delay d.
- $PRFs$: Additionally, the receivers need to know which algorithm is used to derive the hash chain values, the key for the message MAC and the message MAC itself.

Once these values are stored at the receiver, it is able to verify incoming authenticated packets by this sender.

3.3 Broadcasting Authenticated Packets

Once T_0 is reached, a sender is able to broadcast authenticated packets. Using its local time, it calculates the current interval. Since each interval is mapped to one element of the key chain K, there always is exactly one key available for each interval. Using this key in a MAC operation could introduce a repeated use of a single secret key in different cryptographic operations: It was used in generating the next hash chain element and the upcoming MAC operation. Instead of using the key directly, TESLA performs an additional operation: 'Using a pseudo-random function (PRF), f', we construct the one-way function F': $F'(k) = f'_k(1)$. We use F' to derive the key to compute the MAC of messages in each interval. The sender derives the MAC key as follows: $K'_i = F'(K_i)$.' ([PSC+05]) An example for a function f' is a HMAC function like HMAC-SHA256. With this procedure, a sender now is able to generate a secure and unused key K'_i for each interval i.

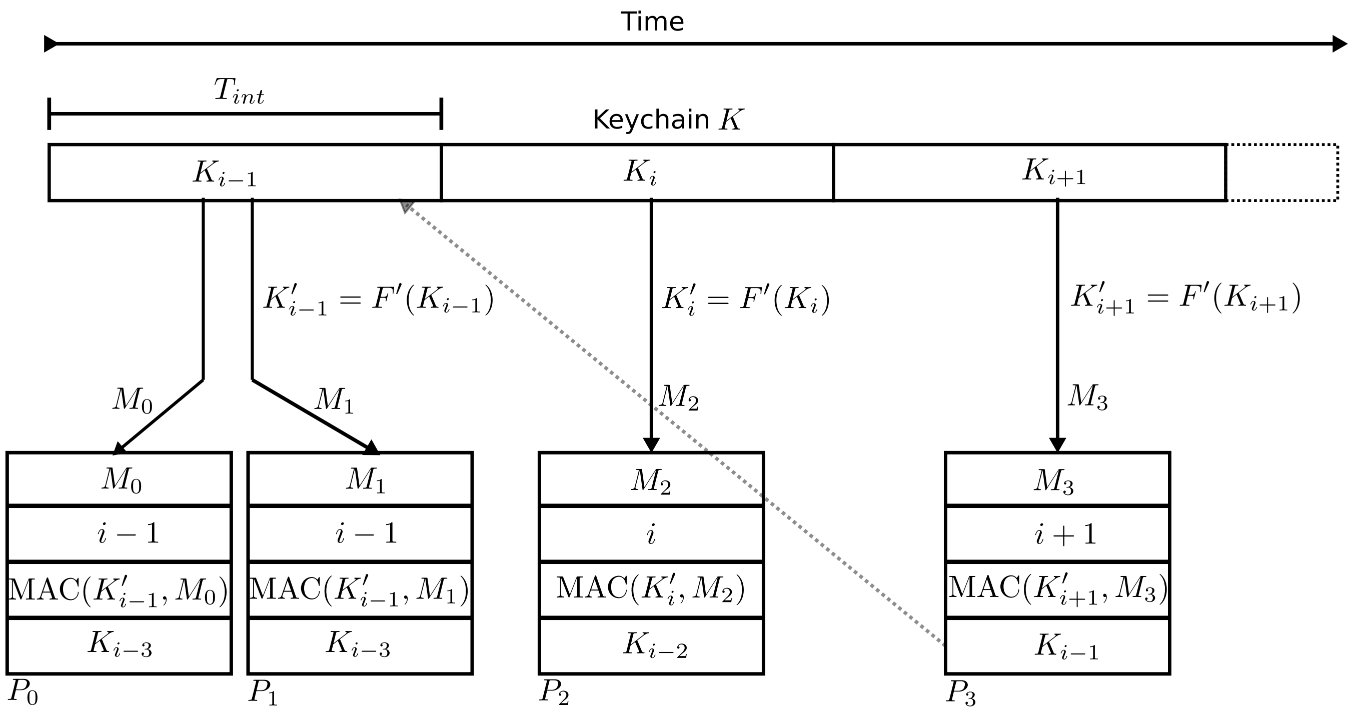
Using K'_i, the sender is able to construct packet P_j for each message M_j, which is sent in interval i, by concatenating the following values:

- M_j: The original message to be be authenticated.
- i: The current interval, derived from the senders time reading and TESLA parameters.
- $\text{MAC}(K'_i, M_j)$: A MAC over M_j, using the secret key K'_i.
- K_{i-d}: The disclosed key. In particular, the secret key used in interval $i-d$. This value is determined by the key disclosure delay d and can be null (For example, a disclosure delay of 2 will imply that all packets sent in interval 0 and 1 will carry no disclosed key).

This also indicates that the sender is not limited by packets per interval, it is rather limited by time. As soon as interval $n-d+1$ is reached, a receiver won't be able to verify incoming packets (See 3.4), which marks the end of transmission. The process of creating authenticated packets is visualized by 3.2, wherein 4 packets over 3 intervals, using a key disclosure delay of 2, are exemplarily sent.

3.4 Verifying Packets

By the end of the last chapter, a sender now is able to broadcast authenticated packets. Following the steps listed below, any bootstrapped receiver is able to verify authenticated packets.



(1)

Figure 3.2: A sender broadcasting 4 packets over 3 intervals with a key disclosure delay of 2

Upon receiving an authenticated packet P_j, containing the message M_j, an interval index i, the MAC $\mathrm{MAC}(K'_i, M_j)$ and a disclosed key K_{i-d}, the receiver first records its local time T. Before any further processing can be done, a receiver has to perform the 'Safe packet test' ([PSC+05]). This test ensures that the sender did not yet disclose the key which was used to authenticate P_j, i.e the key K_i. Using the previously set sender clock drift D_t, the receiver is able to set an upper bound for the time at the sender: $t_j = T + D_t$. The receiver also knows the interval duration T_{int} and the start time T_0, so it can use the formula $x = floor((t_j - T_0)/T_{int})$ to calculate x, the highest possible interval the sender could be in. By verifying $x < i + d$, the packet passed the safe packet test and can be processed further. If the verification fails, other possible malicious senders could have had access to the secret key K_i and have used it to forge an authenticated packet. The packet has to be dropped by the receiver.

The receiver is able to tell whether the packet was safe for further processing. However, it is not able to verify M_j since the secret key K_i used in the MAC generation will be disclosed by a packet in interval $i+d$. It has to buffer $(i, M_j, \mathrm{MAC}(K'_i, M_j))$ and authenticate it later.

The last step is the processing of the disclosed key K_{i-d}. The receiver checks whether it already has a key K_v where $v = i - d$:

A key K_v exists: This implies the receiver already knows the key of interval $i - d$. This can be either by a previously received packet of the same interval i or through the key chain commitment received in the bootstrap phase (e.g. the key K_0). The receiver compares the received K_{i-d} with the stored K_v and drops the packet including the buffered triplet upon mismatch.

A key K_v does not exist: The receiver does not know about a key for interval $i - d$ yet. However, it certainly knows about a previous key K_m where $m < i - d$, either through a previous interval's packets or the initial bootstrap phase (e.g. the key K_0). Utilizing the recursive hash chain property, it verifies that the received key K_{i-d} is indeed a member of the hash chain:

$$K_m = F^{i-d-m}(K_{i-d}) \tag{3.2}$$

If this equation holds true, the disclosed key is a confirmed member of the hash chain and the receiver can continue to verify buffered packet of interval $i - d$. By calculating $K'_{i-d} = F'(K_{i-d})$, the *MACs* of every buffered packet can be re-calculated and compared. If the comparison succeeds, the packet's integrity is verified and can be further processed.

3.5 Packet Loss Support

Depending on the network size, a packet can travel long distances. On this journey, a lot of factors affect the reliable arrival of packets. For example, a packet may be dropped at some point due to too many hops. Even in small scale networks, the outage of a single node can lead to impossible packet routing. As the original TESLA draft specifies, 'Strong robustness to packet loss' ([PSC+05]) is an important property of TESLA. It is achieved due to the properties of the internal hash chain. As we've already seen in Figure 3.1, the hash chain K can be completely reconstructed or build by knowing K_n. It also can be reconstructed partially, from K_0 to K_i, by knowing K_i. With this property applied to TESLA, a receiver is still able to authenticate buffered packets although all packets from the respective key disclosure interval have been lost in transit. All packets from interval $i - d$ can not only be verified by a packet from interval i, but also from any packet of an interval $> i$: As an example, a packet of interval $i + 1$ carries the disclosed key K_{i+1-d}. In the process of checking the hash chain membership of K_{i+1-d}, one also calculates and verifies K_{i-d}, which was not known to the receiver due to packet loss in interval i. This allows the receiver to also verify buffered packets of interval $i - d$ without ever having received the disclosed key K_{i-d}. This is visualized in Figure 3.3 with an example disclosure delay of 2.

3.6 Scalability

In a lot of scenarios, the ability to scale is a key requirement. As seen in 2.4, a simple multicast authentication scheme can lead to an extensive overhead for a large amount of receivers. For TESLA, the following statements regarding scalability can be made:

- The sender does not have to keep a state per receiver. The overall resource utilization is the same for 1 and 1 million receivers.
- The authentication overhead per message does not depend on the amount of receivers, it merely depends on the chosen MAC algorithm and its length.

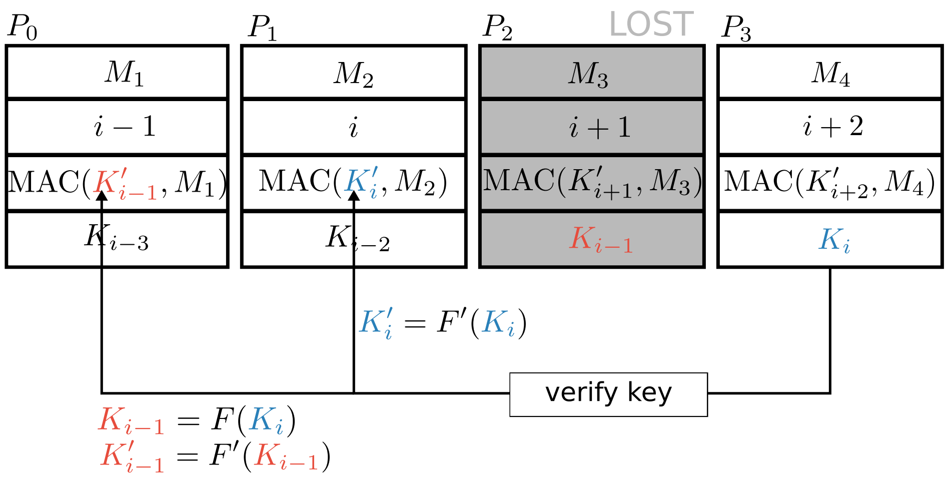


Figure 3.3: Packet Loss Support with a disclosure delay of 2

- TESLA requires every receiver to be bootstrapped. While this bootstrap process is not part of the TESLA protocol, its a strict requirement and its scalability depends on the underlying session setup protocol. For example, an asymmetric based protocol may be inappropriate for small scale receivers. An alternative is given in 4.2, which utilizes a symmetric based bootstrap. However, the sender has to share a unique secret with every possible receiver.

3.7 Denial of Service Attack Vector

Due to its delayed authentication mechanism, TESLA receivers are vulnerable to Denial of Service (DoS) attacks. This can be illustrated using the following example: A receiver R_1 receives a legitimate packet P_j in interval i. Now it is able to craft a fake packet P_{fake} with the following contents:

- M_{fake}: A fake message. Its length has an impact on the victims buffer, so a large message can lead to DoS more quickly.
- i: The interval i
- $\mathrm{MAC}(X_1, X_2)$: A MAC with an arbitrary secret X_1 and payload X_2. However, the length should be equal to the received MAC of P_j. This can also be replaced with a random string.
- K_{i-d}: The disclosed key which was in the original packet P_j.

R_1 is now able to send multiple P_{fake} packets to another receiver R_2. Upon packet receipt, R_2 will buffer the packet as long as K_i is not disclosed yet ('Safe packet test' ([PSC$^+$05])). Depending on the size and amount of fake packets, this can lead to a DoS, since at some point R_2's memory will overflow.

The original draft specifies 2 methods to prevent DoS attacks [PSC$^+$05]. The first method checks whether the source IP is valid and the packet size is not unusual. This, however, requires a controlled network and a rather constant packet size, which is not always achievable. The second option introduces a misordering test:

> Reasonable misordering test: Before the key verification test (Step 3), check whether the disclosed key index $i - d$ of the arriving packet is within g of the previous highest disclosed key index v; thus, for example, $i - d - v \leq g$. g sets the threshold beyond which an out-of-order key index is assumed to be malicious rather than just misordered. Without this test, an attacker could exploit the iterated test in Step 3 to make receivers consume inordinate CPU time checking along the hash chain for what appear to be extremely misordered packets. ([PSC+05])

Another prevention is provided by a TESLA derivative in 4.4, which for one removes the oldest packets once the buffer becomes too large.

In summary, one can take some precautions against DoS attacks, but the delayed authentication buffer always bears a risk of potential packet flooding.

4 Related Work

Multicast authentication is a challenging problem and TESLA is not the only solution available. Efficient Multi-chained Stream Signature (EMSS) is another streaming authentication protocol, which also depends on hash chains and provides similar security services like TESLA does. Furthermore, instead of reinventing the wheel, a lot of TESLA variations have been proposed, providing solutions for different use cases and environments. The next chapters deal with the introduction into this TESLA related work.

4.1 Efficient Multi-chained Stream Signature

With [PCTS00], EMSS was proposed by the same authors who created the TESLA draft [PSC+05]. It's an alternative protocol for streaming multicast authentication, with some minor differences to TESLA. The basic scheme looks like the following:

At the sender, a MAC of each outgoing packet will be calculated and appended to the next outgoing packet. In this way, a linked list of packets is created. So that the receiver is able to verify a packet, the sender periodically sends a special signature packet, which contains the asymmetrically signed MAC of the packet sent before. By verifying this signature packet, the receiver is now able to verify the last packet with its containing MAC. Due to the linked list property, the receiver is able to traverse the previously received packets and verify each packet based on the successor packet.

With the use of an asymmetric algorithm, this protocol also provides non-repudiation for each packet. However, its properties also include a delayed authentication, since the receiver has to wait for the signature packet to arrive, similar to TESLAs disclosure delay. Increasing the number of previous MACs in a packet results in packet loss support with the disadvantage of a larger total size per packet.

4.2 μTESLA

As explained in chapter 3.4, TESLA is not suitable for very small and limited devices due its buffering requirement. μTESLA provides various changes to the protocol, so it can be used in such environments, especially sensor networks [PST+02]:

Symmetric based bootstrap - To support symmetric based bootstrapping, the protocol requires that every receiver shares a unique secret key X between itself and the sender. A receiver is now able to send a nonce N_r to the sender, which in return responds with the payload

$$T_S|K_i|T_i|T_{int}|\delta \tag{4.1}$$

$$\text{MAC}(X, N_r|T_S|K_i|T_i|T_{int}|\delta) \tag{4.2}$$

where

- T_S is the current time of the sender
- K_i is a key of the hash chain, used in a past interval i
- T_i is the start time of interval i
- T_{int} is the interval duration
- δ is the disclosure delay

So a receiver can be fully bootstrapped at any time, using only symmetric algorithms.

Disclosure packets - Instead of sending the disclosure key for the current interval in each packet, μTESLA uses dedicated disclosure packets. This change however could cause worse packet loss handling and increased authentication delay, depending on the amount of disclosure packets per interval.

Restricting the number of senders - To send messages, hash chains are required. These can become very long and thus are very expensive memory-wise, which makes them not applicable to small devices. μTESLA provides two methods to solve this problem. The first one uses the regular, more powerful sender host as a broadcast proxy, requiring an authenticated unicast message from the small node to the sender node. The second method uses the node as the actual broadcaster, utilizing a one-time hash chain element. This chain element has to be delivered to the node in a confidential manner, which can be utilized by using another protocol Secure Network Encryption Protocol (SNEP), also proposed in [PST+02].

4.3 *inf*-TESLA

The original TESLA RFC did not specify what procedure should be performed once the hash chain is exhausted. However, without any further protocol changes, a new session bootstrap is required, which most likely implies the use of asymmetric cryptography. *inf*-TESLA, a modification to the original protocol, tries to solve this problem by implementing *Dual Offset Key Chains* [CAPC16]. The sender has to manage and store two hash chains, instead of just one. Both hash chains have a specific offset to each other, which is also shown in Figure 4.1. To utilize both hash chains, the bootstrap and packet sending/receiving process need

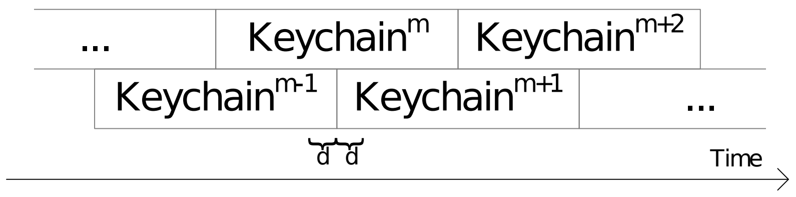


Figure 4.1: Dual Offset Key Chains [CAPC16]

to be adapted: In the bootstrap phase, the sender has to send two hash chain commitments instead of just one, in respect to the currently active hash chains. Furthermore, two keys are now disclosed in each packet and the secret, to generate a packet's MAC, consists of two hash chain values. In the original TESLA protocol, the bootstrap was required to get a commitment to the current hash chain. Now, the question may arise on how *inf*-TESLA solves this problem with multiple hash chains and no intermediate bootstrap messages. This is solved by the above mentioned required offset between hash chains. Thus, it makes sure that at least one active hash chain can be used to verify the first disclosed key of a recently switched second hash chain, allowing the sender to seamlessly switch hash chains.

4.4 TESLA++

Due to the delayed authentication method, DoS attacks can be a quite serious problem for TESLA. The receiver has to store messages and their MACs, so it can verify them later once the secret key for this message has been disclosed. Especially on receivers with a small memory footprint, this procedure allows attackers to overflow a receivers memory very quickly. TESLA++ was first proposed in [SBBP09] and aims to weaken this attack vector by requiring less storage and better DoS handling on behalf of the receiver. Figure 4.2 shows how packets are sent over the wire and how receivers are able to authenticate messages. Instead of sending the triplet (Message M, MAC MAC_S and i), the message M

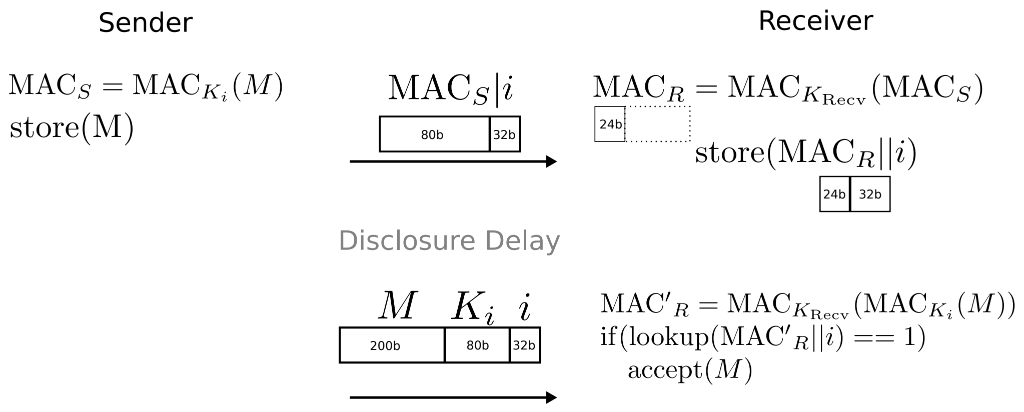

Sender
Receiver
$\text{MAC}_S = \text{MAC}_{K_i}(M)$
store(M)
$\text{MAC}_S|i$
80b
32b
$\text{MAC}_R = \text{MAC}_{K_{\text{Recv}}}(\text{MAC}_S)$
24b
$\text{store}(\text{MAC}_R||i)$
24b
32b
Disclosure Delay
M K_i i
200b
80b
32b
$\text{MAC}'_R = \text{MAC}_{K_{\text{Recv}}}(\text{MAC}_{K_i}(M))$
$\text{if}(\text{lookup}(\text{MAC}'_R||i) == 1)$
$\text{accept}(M)$

Figure 4.2: TESLA++ Communication Overview, adapted and modified from [SBBP09]

will be not sent and instead, is stored. Once the packet arrives, the receiver calculates a new MAC MAC_R over the received MAC MAC_S and shortens it. A secret K_{Recv} is used in this process, which is just known to the receiver itself. Combined with the index i, the MAC_R is stored in memory until the necessary keys are disclosed. Once this disclosure phase starts, the sender sends out a packet containing the actual message M, the disclosed key K_i and the index i. The receiver is now able to verify M by calculating MAC_R once again and checking whether the combination of MAC_R and i exists in the store. Instead of using the MAC_S for the calculation of MAC_R, it will use the disclosed key K_i and the actual message M.

This alternate procedure uses less storage on the receiver side. Especially, its storage is now independent of the message size, since just the MAC and the index are stored. However, the sender now has to store the hash chain and additionally every outgoing message until its disclosure phase.

To provide even more DoS protection, TESLA++ considers the case of low remaining memory on the receiver and provides two procedures:
Remove all shortened MACs, which are older than the last authentic message and 'If more space is needed, the message whose verification is furthest out in the future is discarded' ([SBBP09])

4.5 RIOT TESLA

At the time of writing, another thesis about TESLA is published by a fellow student Christopher Schütze. In [Sch], he likewise analyzes TESLA and provides an implementation with a focus on small scale, IoT devices. The implementation is based on the RIOT operating system. Due to the focus on small scale devices, he proposes different solutions to minimize resource usages. For example, instead of storing the whole sender hash chain in memory, he analyzes the usage of a so called way point system, where just every n-th hash chain value is stored. If required, missing hash chain values can be recalculated from the nearest way point. In chapter 6.5.2, a cross-platform test between the Linux and RIOT TESLA implementations is realized.

4.6 Axiom

Another protocol for securing multicast group communication is Axiom, a DTLS-based approach [TNR17]. Axiom similarly uses symmetric cryptography and provides integrity and even confidentiality to the multicast receivers. Additionally, it allows receivers to securely send a unicast response back to the sender. The basic scheme looks like the following:

Based on a single pre-shared group SA, the sender is able to create authenticated and encrypted multicast messages. Similar, every receiver, which has access to the group SA, is able to authenticate and decrypt the message. Furthermore, utilizing a PRF, receivers are able to derive new keying material based on the group SA, its IP and its unique *SenderID*. Based on this material, it is able to send a secure response to the sender. The sender is able to derive the same keying material used in this response, making the message authenticatable. While this allows a secure group communication, it does not provide end-to-end security between receiver and sender, due to the transparent and not random values used in the key derivation process [PP18]. In summary, while Axiom provides a process for secure unicast responses, it does not provide source authentication and requires a group SA to be present on all participants.

4.7 Identity-Based Signatures

Identity-Based Signatures (IBS), introduced by Adi Shamir in 1984, is also able to provide multicast authentication and even non-repudiation [Sha85]. However, it relies on asymmetric cryptography. The authentication scheme starts with requiring a TTP, which is able to generate a node's private cryptographic parameters based on the node's identity. These parameters have to be sent to the node on a secure, confidential channel. The node is then able to sign messages using its own private parameters. Another node can verify signatures using the public parameters of the TTP and the node's identity.

In another thesis [Mel], Andrian Melnikov designed and evaluated a testbed for IBS. It shows interesting results regarding signing and verifying performance and signature length using different message sizes and two different elliptic curve based algorithms:
Using BLMQ [BLMQ05], rather short signatures are generated, starting from 67 bytes. This is close to TESLA's overhead of approximately 68 bytes per message (Utilizing the SHA256 algorithm). However, BLMQ brings along a rather slow sign and verify process, starting from 592ms and 1465ms on an IoT-Lab M3 node (72 Mhz CPU, 64kB RAM). Melnikov also

had a deeper look at an alternative algorithm, vBNN. It provides a faster sign and verify process (100ms and 300ms), with the cost of larger signatures, starting from 100 bytes. For a detailed statistic and background, refer to [Mel].

IBS is an interesting alternative in cases where non-repudiation is required. However, due to the asymmetric performance requirements, it may not be suited for streaming purposes, depending on the node's capabilities.

4.8 Summary

To summarize this chapter, we analyze the possible usage of these related works in a constrained environment with streaming requirements:
The protocols EMSS and IBS require more computing power due to the asymmetric property. However, if this power is available to all nodes, EMSS seems to be the better choice for a continuous data stream. While it requires a buffer for the delayed authentication, most of the packets are still symmetrically secured, leading to smaller packet sizes and less computational overhead.
The TESLA derivatives μTESLA and TESLA++ add valuable additions to the protocol, which should be included if either devices are very constrained or DoS poses a problem. On the contrary, inf-TESLA solves one of biggest disadvantages of TESLA, the required rekey after the chain is exhausted, very elegantly. The sender already has a larger memory requirement, thus, running *Dual Offset Key Chains* can often be achievable. The addition of another interval and disclosed key to each packet increases the overall size of the traffic (another 36 bytes in case of SHA256 and an interval size of 4 bytes), but allows a source-authenticated multicast data stream of infinite length and a single bootstrap process per receiver.

5 Design

Within the last chapters, a thorough knowledge about ESP and TESLA has been built up. TESLA, as an authentication protocol, can be used to generate ICVs and verify them in a delayed manner. The original draft even specified a way to construct complete, authenticated packets P_j based on a message M_j [PSC+05]:

$$P_j = M_j || i || MAC(K'_i, M_j) || K_{i-d} \tag{5.1}$$

where || denotes concatenation. While this allows source authentication at the receiver, it does not eliminate all attack vectors. For example, an attacker is able to perform a replay attack by duplicating a received packet and forwarding it to other receivers. Instead of adding a sequence number to the TESLA protocol and ICV, the already existing transport protocol ESP is used to provide additional security and features, including a well defined specification and optional encryption. Hence, the following chapter designs a prototype setup, which combines TESLA ICVs and ESP headers to create a source authenticated IPv6-based multicast data stream.

5.1 Architecture

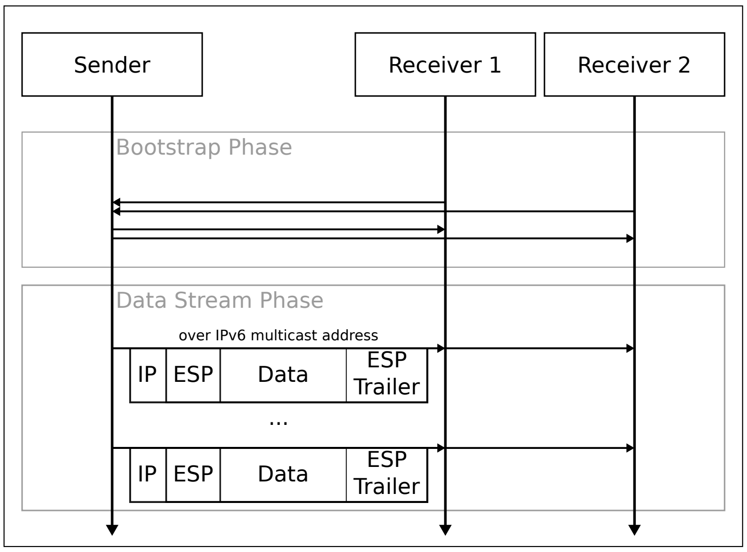


Figure 5.1: Application Prototype Roles and Phases

To generally construct a multicast data stream, the following roles are proposed:

- **Sender**: A single sender node, which will broadcast TESLA authenticated ESP packets in a multicast IPv6 group. In theory, TESLA can be utilized in a true multicast

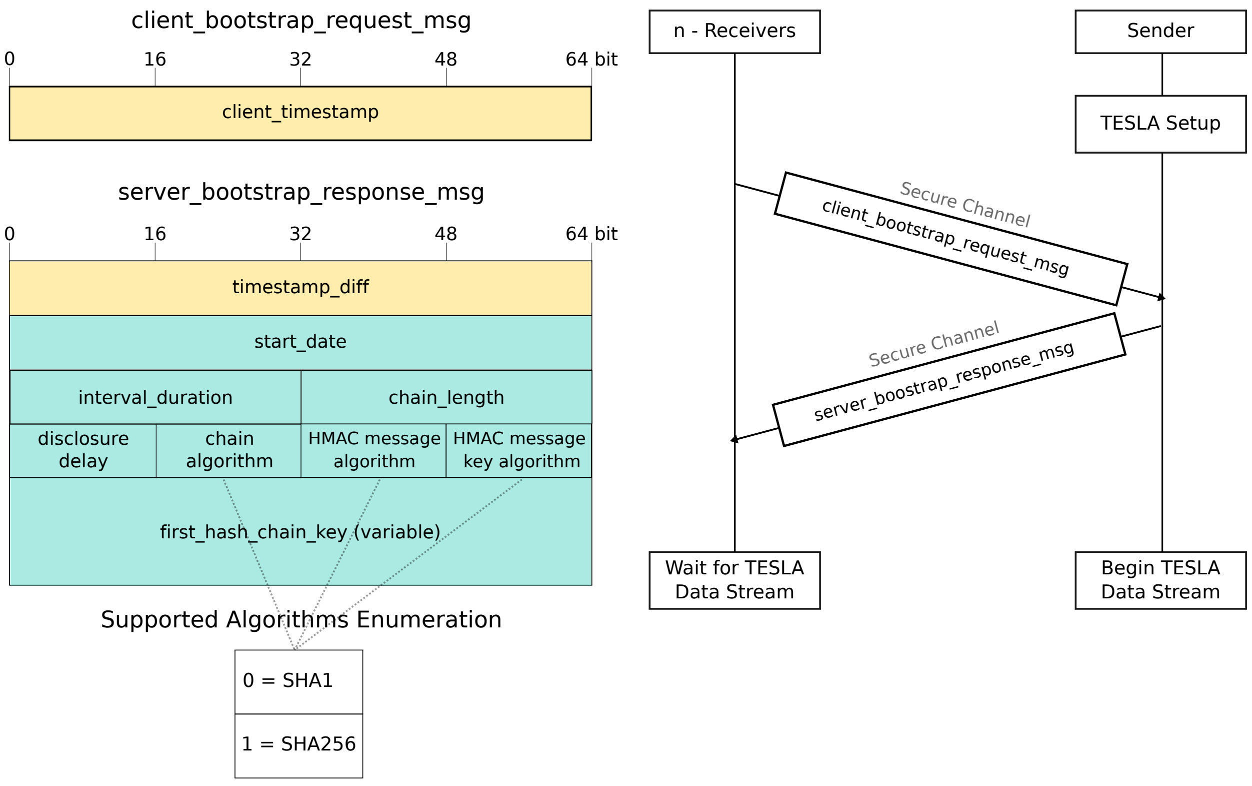


Figure 5.2: TESLA Bootstrap Messages and Exchange

scenario, consisting of multiple senders. However, for simplicity reasons, this design will support only one sender.

- **Receiver**: n receiver nodes, which will listen on an IPv6 multicast address for incoming ESP packets.

Before a sender now is able to send TESLA authenticated packets to multiple receivers, it has to bootstrap every possible receiver with the parameters of the upcoming TESLA stream. Therefor, the application's lifetime is split into two phases: Bootstrap phase and Streaming phase (Figure 5.1). In the following chapters, the network communication and general application behavior for each phase are specified.

5.2 Bootstrapping Phase

Before a receiver application is able to process TESLA ESP packets, it needs to be bootstrapped by an entity, which knows the parameters of the upcoming TESLA broadcast. These parameters also define a TESLA-based SA. In addition to the parameters, the state on the sender, consisting of a hash chain, and the state on the receiver, consisting of a hash chain value and buffered packets, also are part of the SA (The SA struct is not specified and is up for the implementation to decide). With the necessary parameters of chapter 3 in mind, the following bootstrap messages are defined (Figure 5.2):

client_bootstrap_request_msg The message *client_bootstrap_request_msg* is sent by a receiver to the sender, expressing interest in the upcoming TESLA data stream. It contains the following fields:

- **client_timestamp - 8 byte** The current timestamp at the receiver, which is encoded as a unix millisecond timestamp since the Epoch (In the following, *timestamp* refers to a unix epoch timestamp in milliseconds). While this value is not primarily necessary for the TESLA bootstrap, it allows all receivers to loosely time synchronize with the sender. Due to the 8 bytes, the field is large enough for all future timestamps in the next hundred thousands of years. This would not be the case for a 4 byte field, which would be too small already.

server_bootstrap_response_msg As a response to a received *client_bootstrap_request_msg*, the sender can answer with a *server_bootstrap_response_msg*, which contains all necessary values for the upcoming TESLA stream. For the sake of simplicity, the message does not contain any form of negative feedback, i.e it's not used to deny access to the data stream. In detail, the following values are defined:

- **timestamp_diff - 8 bytes** The difference between the current sender timestamp and the received client timestamp, which allows receivers to loosely time synchronize themselves. This procedure slightly differs from the original procedure in 3.2. Instead of sending both timestamps back to the receiver, the difference is calculated directly at the sender, leading to a shorter message size with no additional drawbacks.

- **start_date - 8 bytes** The start timestamp of the TESLA transmission.

- **interval_duration - 4 bytes** The interval duration in milliseconds. unsigned 4 bytes allow a maximum value of 4.294.967.295, which satisfies most common use cases.

- **chain_length - 4 bytes** The overall hash chain length.

- **disclosure_delay - 2 bytes** The disclosure delay, with a maximum of 65.535

- **chain_, hmac_message_ and hmac_message_key_algorithm - 2 bytes each** To support different algorithms, the 3 fields define the utilized algorithm. For the prototype, two different values can be declared: *1 = SHA256* and *0 = SHA1*. While a 1 byte field each would also suffice, the size of 2 bytes was chosen to better support memory alignment on the hosts.

- **first_key - variable bytes** The first key of the hash chain. The length is based on the value of the *chain_algorithm* field. In the case of this specification, it's either 20 (SHA1) or 32 (SHA256) bytes long.

With the above messages defined, a receiver now can be bootstrapped by a sender. However, as specified in the original draft [PSC$^+$05], both messages have to be transferred on a secure channel to allow no man-in-the-middle attack vector. The next chapters show two different solutions using key exchange protocols of the IPsec suite, IKEv2 and G-IKEv2.

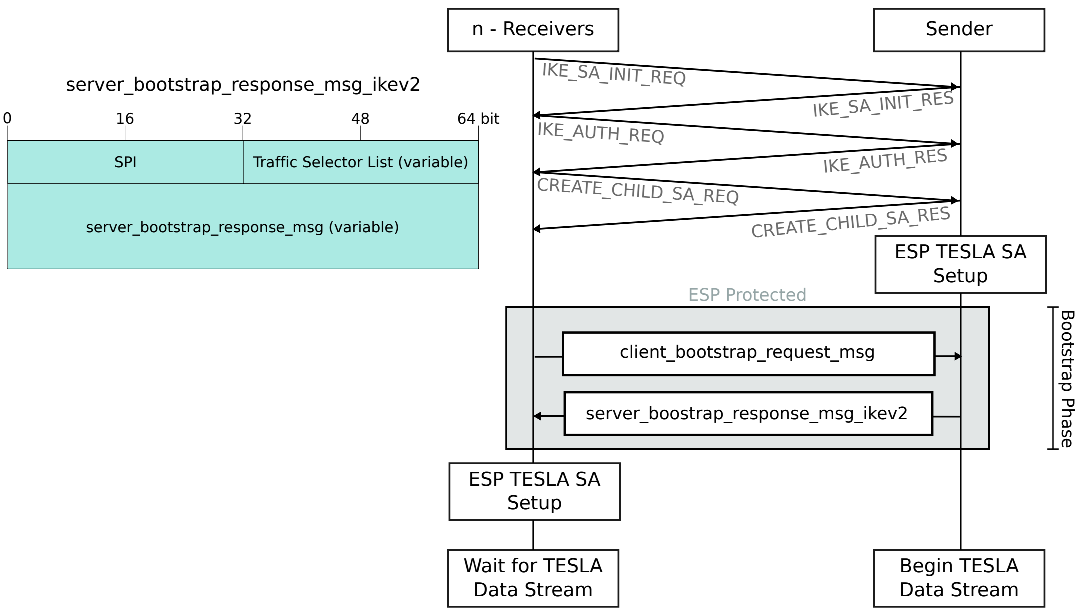


Figure 5.3: IKEv2-based Bootstrap

5.2.1 IKEv2 and ESP

IKEv2, as specified in [KHN+14], is able to create child SAs as soon as the Internet Key Exchange (IKE) SA has been negotiated and set. However, the specification bares one problem for the creation of TESLA ESP child SAs: It is defined, that all further keying material for child SAs are derived from nonces and Diffie-Helmann values from the initial *ike* message exchange. In more detail, as soon as new keying material is needed, both values are fed to a special PRF generator, which continuously is able to create new secret keys. In the case of a TESLA SA, this is not feasible. A shared state would be required to deliver the same TESLA ESP SA to every receiver, which is not specified in IKEv2. To avoid this problem, the following approach is proposed:
Before the TESLA bootstrap is initiated, receiver and sender negotiate a normal ESP secured tunnel via IKEv2 (Figure 5.3), utilizing standard authentication and encryption algorithms (For example. *AUTH_HMAC_SHA2_256_128* and *ENCR_AES_CBC*). Using this secure tunnel, the bootstrap messages can be exchanged in a secure manner. This approach, although, has the disadvantage of having to manually create and manage the TESLA ESP SAs at the application level on both sides. Additionally, the *server_bootstrap_response_msg* has to be adapted: A SPI and possible TSs have to be included, since the TESLA ESP SA requires those values as well (The structure of these fields are not specified, but can be adopted from [KHN+14]). On the other hand, it brings the advantage of needless protocol modifications for IKEv2 and a end-to-end secure communication channel between both participants, which can for example be used for unicast responses from the receiver.

5.2.2 Group IKEv2

To circumvent the IKEv2 issues with TESLA ESP SAs, a solution utilizing the G-IKEv2 protocol, which is especially designed to establish secure group communication channels, is

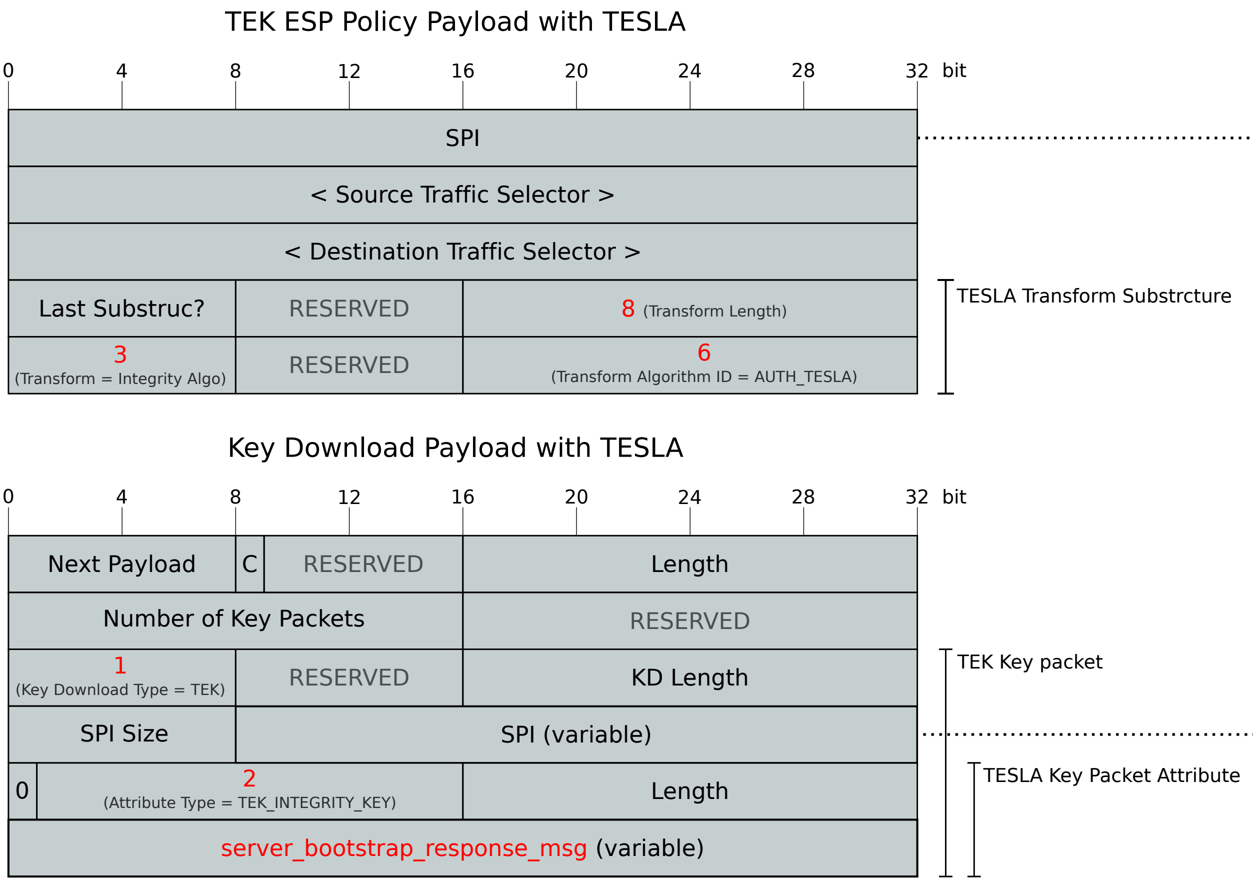

TEK ESP Policy Payload with TESLA
0
4
8
12
16
20
24
28
32 bit
SPI
< Source Traffic Selector >
< Destination Traffic Selector >
Last Substruc?
RESERVED
8 (Transform Length)
3 (Transform = Integrity Algo)
RESERVED
6 (Transform Algorithm ID = AUTH_TESLA)
TESLA Transform Substrcture
Key Download Payload with TESLA
Next Payload
C
RESERVED
Length
Number of Key Packets
RESERVED
1 (Key Download Type = TEK)
RESERVED
KD Length
TEK Key packet
SPI Size
SPI (variable)
0
2 (Attribute Type = TEK_INTEGRITY_KEY)
Length
TESLA Key Packet Attribute
server_bootstrap_response_msg (variable)

Figure 5.4: G-IKEv2 *GSA_AUTH* Response Headers with TESLA Support

presented.
One of the important differences between G-IKEv2 and IKEv2 is the way how the keying material for new SAs is generated. While IKEv2 relies on previously exchanged and random values to generate new keys, G-IKEv2 allows to simply download existing group keying material. With this change, every receiver is able to receive the same TESLA parameters for its local ESP SA. With chapter 2.5.7 in mind, the following procedures add TESLA parameter support for underlying ESP SAs:

- The TEK payload, which is part of the GSA payload and included in the *GSA_AUTH* response, needs to have support for the TESLA authentication algorithm. In detail, the *Transform Substructure List* field of the TEK ESP policy needs to have a transform entry, specifying the use of TESLA for integrity checks (As shown in Figure 5.4). To achieve this support, the next available number/ID 6 in the 'Transform Type 3 (Integrity Algorithm)' ([KHN$^+$14]) needs to be assigned to TESLA. At this point, the transform entry may also contain some of the TESLA parameters as transform attributes, like the utilized algorithms. Even so, for simplicity, all TESLA parameters are combined in the KD payload, which is discussed in the next paragraph.

- By specifying the *AUTH_TESLA* ID in the TEK payload, a receiver knows that

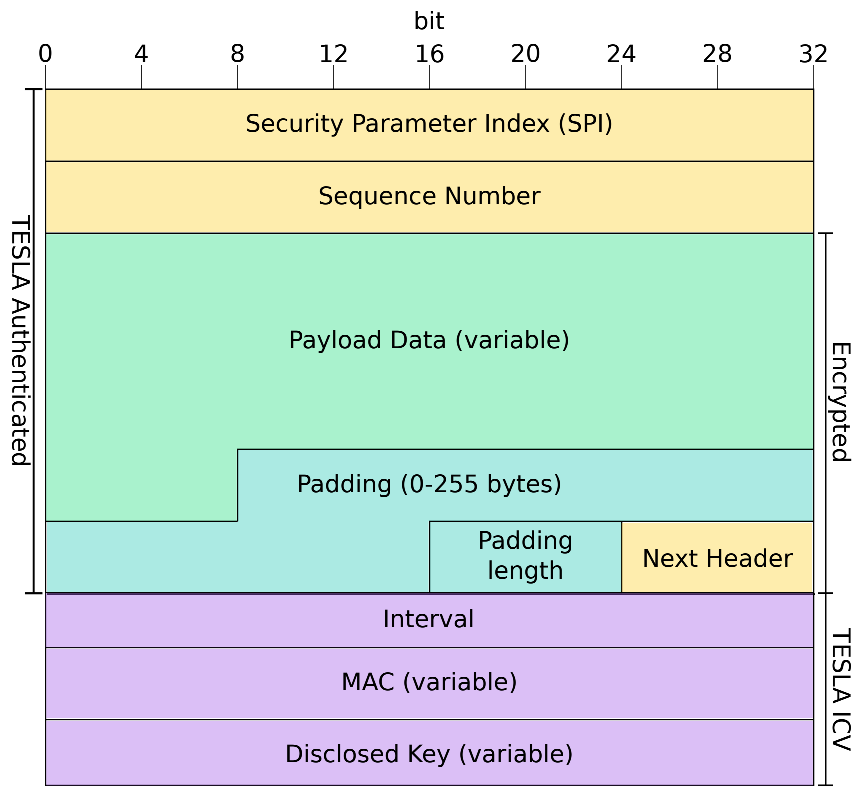


Figure 5.5: TESLA ICV header and it's use in ESP

TESLA is utilized. However, it did not receive any parameters yet. While the parameters are also sent to the receiver in the *GSA_AUTH* response, they are inside of the KD payload. The KD payload can consist of multiple key packets of different types. Important for the TESLA parameters is the key packet type TEK, since it contains keying material for the actual secure data tunnel. By adding a key attribute of the type *TEK_INTEGRITY_KEY*, containing the previously defined *server_bootstrap_response_msg*, to the TEK key packet, the parameters can be transferred without defining new protocol numbers (As shown in Figure 5.4).

With the above design, a successful TESLA ESP SA can be setup. All advantages of the protocol can be used, including a specified rekeying process once the TESLA parameters are obsolete due the hash chain being exhausted. Despite that, the design has one flaw: G-IKEv2 uses the same *IKE_SA_INIT* as present in IKEv2 and does not offer any functionalities to let the receiver send custom data alongside it. A suchlike receiver is not able to send the *client_bootstrap_request_msg* with it, making an in-process loosely time synchronization not possible. An out of band time synchronization process has to be chosen.

5.3 Streaming Phase

After the bootstrap phase has ended, the sender and all receivers have established a TESLA ESP SA, containing the correct TESLA parameters and necessary IPsec parameters. As soon as the start time is reached, the sender can construct and send ESP packets, protected with a TESLA ICV, over the specified IPv6 multicast address. Since a TESLA ICV consists of multiple values and the original draft did not specify any header format, the following header format is introduced for it (As shown in Figure 5.5):

- **Interval - 4 bytes**: The interval in which the sender was, when it sent the packet. Unsigned 4 bytes allow a maximum value of 4.294.967.295, which satisfies most common use cases.
- **Payload MAC - variable**: The MAC of the payload, as specified by chapter 2.5.2. The length can be omitted since the client is able to derive the length of the payload MAC using the TESLA ESP SA.
- **Disclosed Key - variable**: The disclosed key of a previous interval. Similar to the above field, the length can also be derived from the TESLA ESP SA. If the packet was sent in the first d intervals, where d is the key disclosure delay, it has to be filled with zeros by the sender.

The sender is allowed to send an arbitrary amount of packets, as long as they are sent between the start and end time of the TESLA transmission. However, if a certain packet rate per interval is reached, receivers may drop additional packets due to limited buffering capabilities. Once a transmission phase has ended, i.e the underlying hash chain is exhausted and a rekeying operation is required. The sender calculates a new hash chain and possibly new TESLA parameters and updates its TESLA ESP SA. Since all participants are able to tell when this moment arrives, receivers can request a rekey or the server automatically keeps track of all receivers and rekeys them automatically, depending on the implementation. However, a simple rekey introduces a problem: All packets of the last d intervals can not be authenticated by the receiver, since the disclosed keys were never transmitted (Figure 5.6). The new transmission will not carry any disclosed keys in the first d intervals. To avoid this limitation, two SAs need to be used simultaneously. In detail, once the first transmission ends, the TESLA ESP SA will not be swapped, but rather a second one will be introduced. Then, all packets of the second transmission, which are sent in the first d intervals, will contain the remaining disclosed keys of the first transmission (This assumes that the value of d will not shrink between transmissions). As soon as all remaining keys have been sent/received, the first TESLA ESP SA can be deleted and each packet contains the disclosed keys of the second transmission as usual. (Figure 5.7)

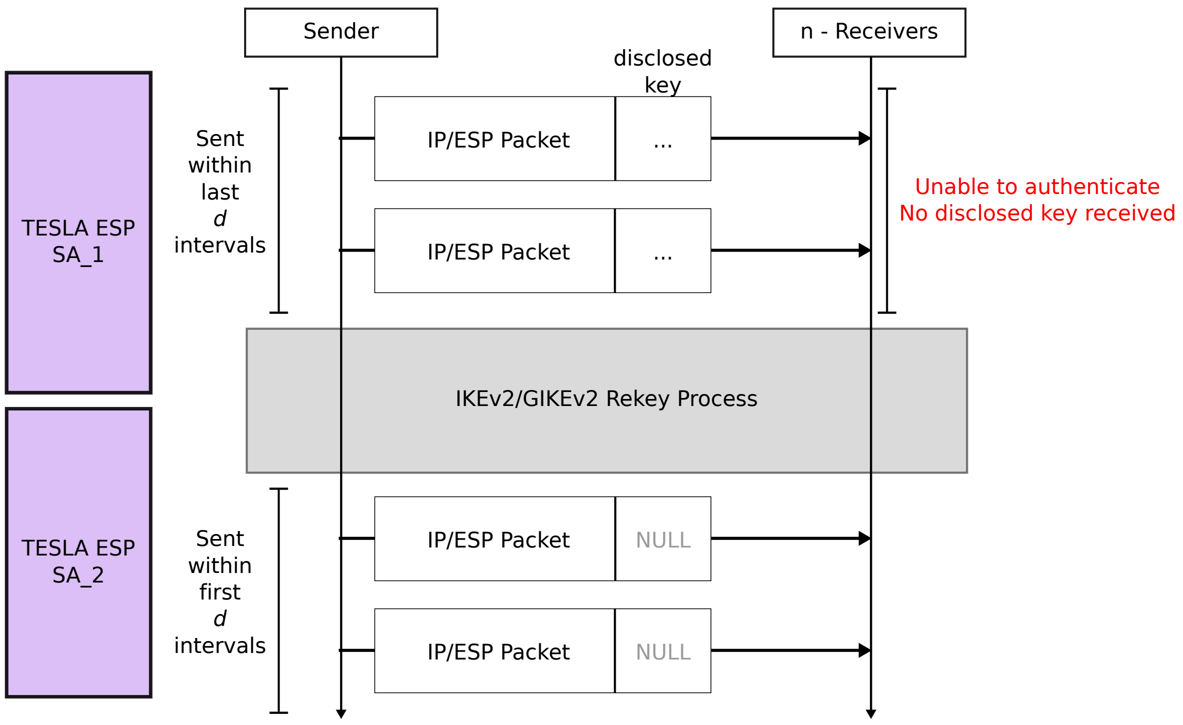


Figure 5.6: Simple TESLA Rekeying process

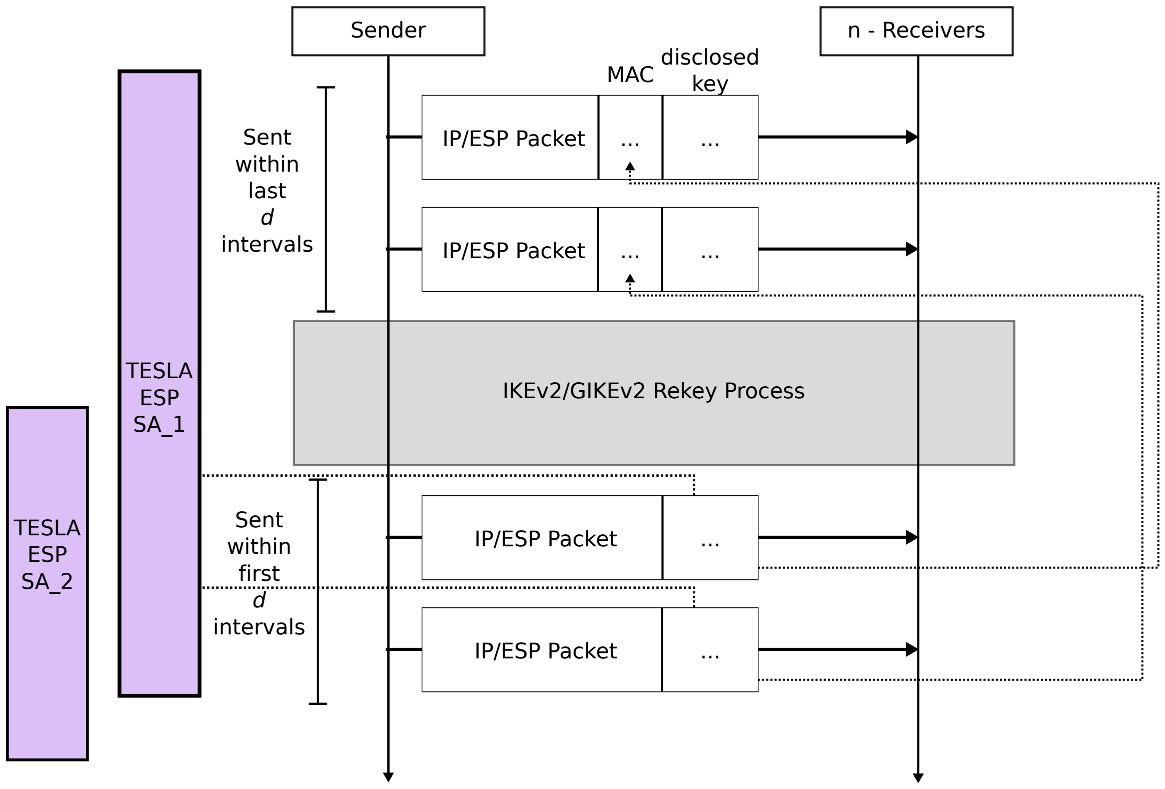


Figure 5.7: Advanced TESLA Rekeying process - No packet loss

6 Implementation

The last chapter designed a process for creating a TESLA-based multicast data stream, utilizing protocols of the IPsec suite. Based on this, the upcoming chapter presents an implementation written in the C programming language. The project is split into **3 sub-projects** (Figure 6.1): The first component is a user space **TESLA Library**, which allows to generate and verify TESLA ICVs according to a specified TESLA configuration. Next up, a user space **ESP Library** is implemented. It allows applications to manage a SAD and create and process ESP headers. The last component is the actual **Test Application**. It utilizes both libraries to bootstrap receivers and create an IPv6 multicast broadcast, on which TESLA protected ESP headers are sent and received by multiple receivers. While the libraries are distributed via statically linked C libraries, the *Test Application* provides a sender and receiver binary. The chapter starts with an introduction into the Linux platform and the project's single dependency, OpenSSL. It then continues with a more detailed look on some important and platform specific parts of each component.

6.1 Platform and Dependencies

The above introduction mentioned each component of the project. However, the mentioned libraries and application require a platform to run on. While there are multiple different operating systems available, the following chapters introduce Linux and the reasons why it is the selected platform for the implementation. The only dependency, OpenSSL, for one a general-purpose cryptography library, is presented in the last part.

6.1.1 Linux

In 1991, Linus Torvalds released an operating system kernel called Linux. By now, it's probably one of the most used kernels on the world, operating on hardware of completely different sizes and field of industry. Additionally, Linux is distributed under the GPL2 license, making it Free Open-Source Software (FOSS). This allows vendors and operating system developers to modify and adapt the source code to their needs, allowing custom, Linux based kernels to be released to the community. While Linux per se is just a Kernel, the name is also often used to describe a group of operating systems based on the Linux Kernel, i.e Linux distributions. Besides its use in mobiles phones and server systems, according to [lin], Linux is also the leading operating system used in IoT projects.

Linux is one of the most widespread operating systems, allows a straightforward low-level network access utilizing the POSIX standard and works well with the programming language C. This makes it a great choice as a platform for the above architecture.

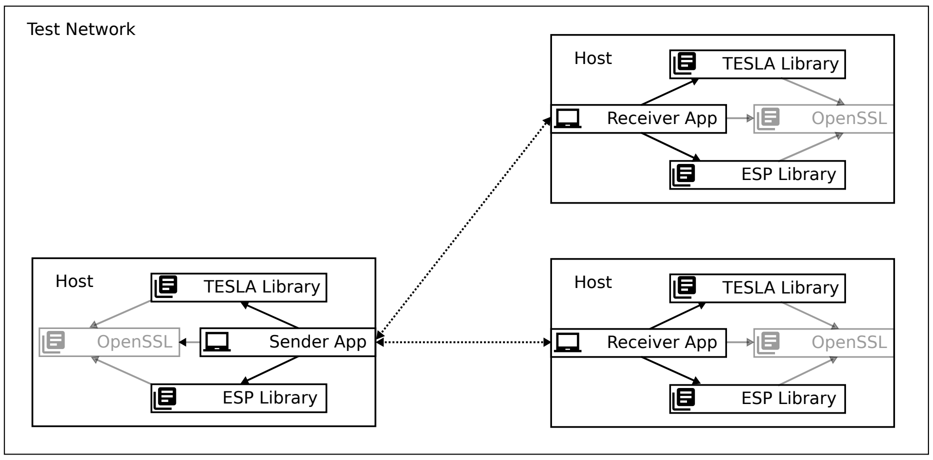


Figure 6.1: Implementation Components and Architecture

6.1.2 OpenSSL

TESLA and ESP both require cryptographic functions to work properly. In the case of TESLA, hash functions, like SHA256, and MAC functions, like HMAC, are needed to generate the hash chain and authenticate/verify packets. ESP likewise depends on MAC functions, but furthermore requires encryption algorithms. Instead of implementing the above functions in the project directly, a dependency to OpenSSL is added. 'OpenSSL is a robust, commercial-grade, and full-featured toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. It is also a general-purpose cryptography library.' ([Opec]) Adopting this library has multiple advantages over an own implementation:

- The library is used in a widespread manner. Thus, more bugs are detected and fixed faster.
- Various different cryptographic algorithms have been implemented. Supporting different algorithms can be solved by exchanging function names instead of writing new ones.
- The code is thoroughly tested and its memory efficiency and performance is continuously improved.

OpenSSL is available on most common Linux distributions and features a command line interface and a C API.

A considered alternative is the Linux *Crypto-API*, introduced in kernel version 2.5.45 [Ekl]. The main difference shows up in the running context: The *Crypto-API* is running in the kernel space, which is a privileged space for the kernel itself and its kernel modules. This enables a faster calculation of ciphers and suchlike. Additionally, since the *Crypto-API* is embedded into the kernel, no additional installation is required. However, due to a confusing documentation and a more complex usage, the integration of OpenSSL is the preferred choice for this project.

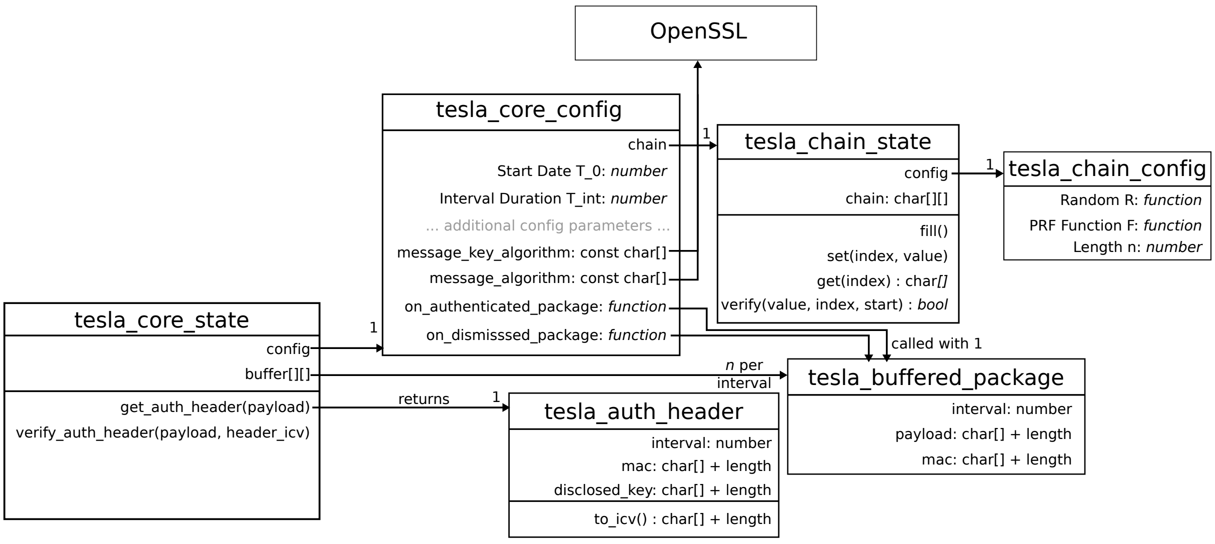


Figure 6.2: Structure of the TESLA Library (Some names may be shortened for brevity)

6.2 TESLA Library

The TESLA library provides applications with functions to create and verify TESLA ICVs. For the sake of separation of concerns and an easier public API, the library is split into the following modules (Figure 6.2):

- **tesla_core.{c | h}**: This module contains the main data structures and operations for the protocol. This includes a configuration struct *tesla_core_config* for defining TESLA parameters, like the disclosed key length and used algorithms. Furthermore, a state struct *tesla_core_state* is provided. This struct is used to manage internal state and acts as a pointer to a configured and initialized TESLA instance.
- **tesla_chain.{c | h}**: A module providing a operations to create and fill hash chains. It follows the same approach as the *tesla_core* module, it provides a configuration and state struct: *tesla_chain_config* and *tesla_chain_state*.

6.2.1 Configuration and Initialization

Before the TESLA operations can be used, the application needs to configure and create a hash chain TESLA instance.

Hash Chain

```
typedef struct tesla_chain_config {
  void (*rand_func)(unsigned char* output);
  int rand_len;
  unsigned char* (*hash_func)(const unsigned char* input, size_t n,
    unsigned char* output);
  const char* hash_func_name;
  int hash_length;
  unsigned long chain_length;
} tesla_chain_config;

typedef struct tesla_chain_state {
  tesla_chain_config* config;
  unsigned char** chain;
} tesla_chain_state;
```

The hash chain configuration is specified by the above struct and covers all possible configuration values. The *rand_func* and *rand_len* provide a dynamic way of defining the seed of the hash chain. It is left to the application to define this function, so different random generators can be used. For example, an application may use the OpenSSL function *RAND_bytes(output, len)* or a hardware random generator.

Furthermore, the actual hash operation is also configured as a function *hash_func*. The signature of this function is compatible with the hash algorithm API of OpenSSL, which allows applications to directly set the random function to an OpenSSL function. The overall chain length is set by *chain_length*, which is stored in an *unsigned long*, allowing a maximum length of $4,294,967,295$.

Based on a configuration, a reference to a *tesla_chain_state* struct is used to keep track of the actual state, i.e the list of hash chain values. For the storage of the hash chain values, a simple array has been chosen. This allows the TESLA implementation a fast index-based access to each value.

```
tesla_chain_state *chain;
static tesla_chain_config chain_config = { ... };

tesla_chain_init(&chain, &chain_config);
tesla_chain_fill(chain);
```

The above shows how an actual chain state is created. Behind the scenes, the *tesla_chain_init* call allocates memory for the *tesla_chain_state* struct, based on the passed configuration. However, actual values are being filled in by the call to *tesla_chain_fill*. In a TESLA context, this allows senders to fill the chain and receivers to just allocate the necessary amount of memory.

TESLA Core

```
typedef struct tesla_core_config {
  tesla_chain_state* tesla_chain;
  unsigned long long start_date;
  unsigned long interval_duration;
  unsigned int disclosure_delay;
  int clock_lag; // receiver only

  const char* message_algorithm;
  const char* message_key_algorithm;

  TESLA_BUFFER_METHOD buffer_method;
  unsigned int buffer_per_interval;

  void (*on_authenticated_package)(tesla_buffered_package* package); // receivers
  void (*on_dismissed_package)(tesla_buffered_package* package, int reason); // receivers
  // some fields omitted for brevity
} tesla_core_config;

typedef struct tesla_core_state {
  tesla_core_config* config;
  unsigned int* buffer_counter;
  tesla_buffered_package*** buffer;

  // some fields omitted for brevity
} tesla_core_state;
```

A TESLA instance can be configured in a similar manner. The first part of the configuration covers timing related TESLA parameters and a configured hash chain instance. Regarding algorithms, the library internally forces the use of HMAC, but allows applications to configure the underlying hash operation with OpenSSL's *EVP_MD* API: The application just has to pass a name for the used algorithm, which internally is then fetched via the *EVP_get_digestbyname* function. Following up, the way how the library buffers packets can be configured, which is described in more detail in the upcoming chapter 6.2.3.

The state itself contains a reference to the configuration and a 2 dimensional array of packet references. This array is used as a packet buffer. In detail, it stores a list of received packet references per interval. To keep track of the number of received packets per interval, an array of counters is also stored in the state. The allocation of those fields take place once a call to *tesla_core_init(state**, config*)* is made.

6.2.2 Sign and Verify Operations

Once a TESLA state is created and allocated, it can be used to sign and verify arbitrary data. The creation of a new ICV can be achieved with the function *tesla_core_get_auth_header*:

```
int tesla_core_get_auth_header(tesla_core_state* state, unsigned char* payload,
  unsigned int payload_len, tesla_auth_header** header) {

  tesla_core_config* config = state->config;
  tesla_chain_config* chain_config = config->tesla_chain->config;

  tesla_auth_header_init(header, config->message_algorithm_length,
    chain_config->hash_length); // allocate memory for TESLA header

  (*header)->interval = tesla_core_get_curr_interval(state);
  unsigned char* key = malloc(
    sizeof(unsigned char) * config->message_key_algorithm_length);
  tesla_core_get_curr_message_key(state, key, NULL); // derive key

  HMAC(config->message_algorithm, key, config->message_key_algorithm_length,
    payload, payload_len, (*header)->mac, &(*header)->mac_len);

  unsigned char* disclosed_key = NULL; // attach disclosed key, if available
  if(tesla_core_get_curr_disclosed_key(state, &disclosed_key) == 0) {
    memcpy((*header)->disclosed_key, disclosed_key, chain_config->hash_length);
  }

  free(key);
  return 0;
}
```

Based on the input payload and its length, the function allocates and fills a passed *tesla_auth_header* struct, according to the original TESLA specification. It uses OpenSSL's HMAC function in combination with the config's *EVP_MD* algorithm to create the packet's MAC. Additionally, it checks whether a key is disclosed in the current interval. The output struct then contains the current interval, the MAC with its length and an optional disclosed key with its length. However, this struct is not meant to be serialized and sent over a network. A call to the function *tesla_auth_header_to_icv(header*, output**)* would strip the lengths and write a serialized version into the output buffer (The output is in the format of the designed TESLA ICV of chapter 5.3).

Since the verify operation contains multiple steps, it will be split and explained in 4 parts:

```
int tesla_core_verify(tesla_core_state* state, unsigned char* payload,
  unsigned int payload_len, unsigned char* auth_header, unsigned int auth_header_len) {

  tesla_auth_header* header;
  tesla_auth_header_init(&header, config->message_algorithm_length,
    chain_config->hash_length);

  tesla_auth_header_parse(auth_header, auth_header_len, header);
  // continue
```

Listing 6.1: 1. Part of Verification - Parse ICV header

The verify method is commonly used at the receiver side. Thus, instead of requiring an already parsed *tesla_auth_header* struct, an ICV format is expected in *auth_header*. With the help of the configuration values, the first step then tries to parse the ICV format into a well-known header format, failing if specific lengths do not match.

```
long long curr_time = get_time_ms();
long long upper_bound = curr_time + config->clock_lag;
unsigned long possible_interval =
  floor((upper_bound - config->start_date) / config->interval_duration);

if(possible_interval >= header->interval + config->disclosure_delay) {
  return -2;
}
// continue
```

Listing 6.2: 2. Part of Verification - Upper Bound Check

Next is an upper-bound check as described in 3.4. Since the library uses millisecond timestamps as a time unit, all time related variables have to be at least of type *long long*. This ensures, that at least 64-bit are used to store the variable, which allows values up to $9,223,372,036,854,775,807$. A variable of type long, i.e 32-bit, would not be sufficient: Any current millisecond unix timestamp (e.g. $1,540,538,952,736$) is already larger than the available space ($2,147,483,647$), which is also the case for *unsigned long* ($4,294,967,295$).

An important part of the verifying process is handling the received disclosed key, which can be divided into 3 different cases:

- **No disclosed key expected**: In the first d intervals, where d is the disclosure delay, no disclosed keys are expected.

- **Disclosed key already received**: When multiple packets are sent in the same interval, they carry the same disclosed key. If a disclosed key was already received, the new one will be compared to the old one and the packet is only kept if they match. Comparing cryptographic values with standard functions, like *memcmp*, can open the application to timing attacks. Therefore, the dependency to OpenSSL is utilized and its constant time comparison function *CRYPTO_memcmp* is called instead.

```c
unsigned long disclosed_interval = header->interval - config->disclosure_delay;
if(state->latest_disclosed_received >= disclosed_interval) {
  // We already received the key, let's compare and check validity
  unsigned char* received_key =
    tesla_chain_get(config->tesla_chain, disclosed_interval);

  if(CRYPTO_memcmp(received_key,
    header->disclosed_key, chain_config->hash_length) != 0) {

    return -3;
  };
}
```

Listing 6.3: 3.a Part of Verification - Disclosed key already received

- **New disclosed key**: Upon receiving a packet from a new interval, it contains a new disclosed key. By storing the interval of the last received disclosed key, we're able to efficiently check whether the disclosed key is part of the key chain. If the key is valid, it will be saved into the key chain at the specific index. together with re-calculated keys of previous intervals. Due to time constraints, the implementation does not clear old received keys. In fact, it currently requires the receiver to also have a hash chain allocated, with the size of the sender's one. The last step of this part is the call to *tesla_core_unbuffer_packages*, which will check the MACs of all buffered packets since the last disclosed interval to the newly *disclosed_interval* and calls the callback functions if applicable.

```c
else if(state->latest_disclosed_received < disclosed_interval) {
  int good_key = tesla_chain_check(config->tesla_chain,
    header->disclosed_key, header->disclosed_key_len,
    disclosed_interval, state->latest_disclosed_received);
  if(good_key == 0) {
    tesla_chain_set(config->tesla_chain, disclosed_interval, header->disclosed_key);
    tesla_chain_fill_between(config->tesla_chain, state->latest_disclosed_received,
      disclosed_interval);

    tesla_core_unbuffer_packages(state, disclosed_interval, state->latest_disclosed_receivec
    state->latest_disclosed_received = disclosed_interval;
  }
}
```

Listing 6.4: 3.b Part of Verification - New disclosed key

6.2.3 Buffering packets

As mentioned in the previous configuration chapter, the library's packet buffer can be configured via the *buffer_method* configuration field in two ways:

- **TESLA_BUFFER_STATIC**: The packet buffer will be initialized in a static manner. In detail, the buffer allows every interval to receive a maximum of *buffer_per_interval* packets. Once the buffer of an interval is full, no more packets will be buffered and are lost. This approach is convenient for applications, which expect a constant number of packets per interval throughout the entire stream.
- **TESLA_BUFFER_GROW**: The packet buffer will also be initialized in a static manner, supporting up to *buffer_per_interval* packets per interval. However, once a single interval buffer is full, it will dynamically double the size of all interval buffers. This approach does not lose any packets, but can lead to a high memory allocation as soon as a single interval contains a lot of packets.

```
if(index + 1 >= config->buffer_per_interval &&
  config->buffer_method == TESLA_BUFFER_GROW) {

  config->buffer_per_interval *= 2;
  for(int i = 0; i < state->max_intervals; i++) {
    state->buffer[i] = realloc(state->buffer[i],
      sizeof(tesla_buffered_package*) * config->buffer_per_interval);
  }
}
```

Listing 6.5: Syllabus of the packet buffering grow logic

6.3 ESP Library

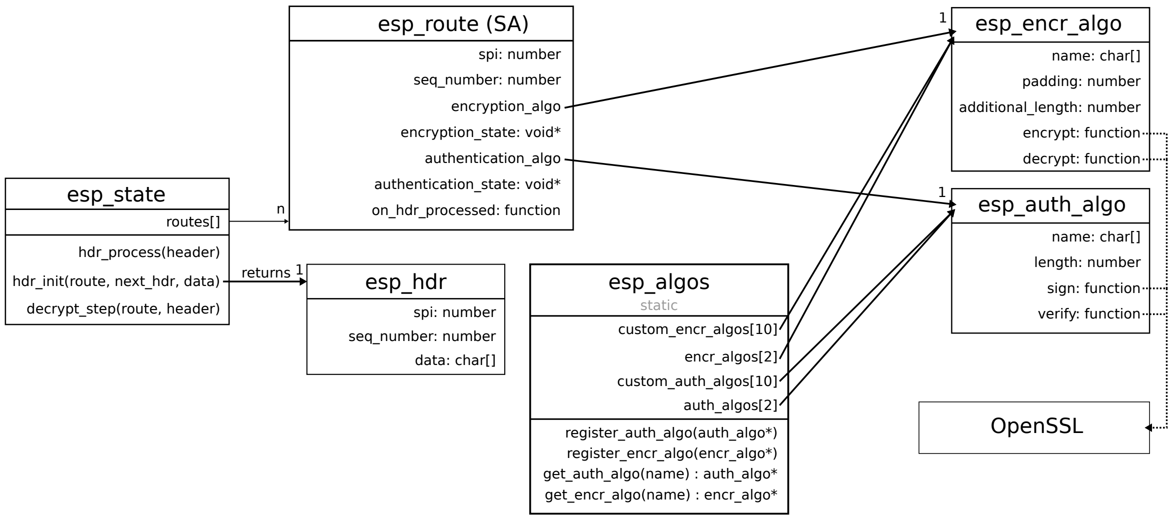


Figure 6.3: Structure of the ESP Library (Some names may be shortened for brevity)

The ESP library provides applications with functions to create and process ESP packets. It's not responsible for network operations, nor does it implement all required algorithms specified in [WMM$^+$17]. However, it allows applications to register their own encryption and authentication algorithms. It is structured in the following way (Figure 6.3):

- **esp.{c | h}**: A module providing the main entry point for processing and generating ESP headers. Similar to the TESLA library, it consists of a state struct *esp_state*, which again holds a reference to a configuration. In this case, the configuration is just a list of *esp_route* structs, forming the SAD.
- **esp_algos.{c | h}**: A module containing the encryption and authentication algorithm abstraction and registration logic. In total, 4 algorithms have been implemented with the help of OpenSSL and with [WMM+17] in mind:
  ENCR_NULL, *ENCR_AES_CBC_128*, *AUTH_NONE*, *AUTH_HMAC_SHA1_96*.

6.3.1 Configuration and Initialization

```
typedef struct esp_route {
  ESP_TYPE type;
  unsigned int spi;
  unsigned int seq_number;
  esp_encr_algo* encr_algo;
  void* encr_data;
  esp_auth_algo* auth_algo;
  void* auth_data;
  void (*on_hdr_processed)(struct esp_route* route, int nxt_hdr,
    unsigned char* data, unsigned int data_len, void* add_data);
} esp_route;

typedef struct esp_state {
  esp_route** routes;
  unsigned int routes_len;
} esp_state;
```

Listing 6.6: ESP state and route structs

The setup and configuration is similar to the TESLA library. Each route consists of a reference to one encryption and one authentication struct. Since most algorithms also require some state, *encr_data* and *auth_data* are also supplied. By making both states of type *void**, not only static data is supported, but also references to more complex structs. By supplying multiple *esp_routes* to the *esp_state_init* call, an *esp_state* is allocated and created.

6.3.2 Create and Process Packets

```
// esp_hdr_create(route*, esp_hdr**, next_hdr, data*, data_len)
unsigned int add_space = route->encr_algo->additional_space;
(*hdr) = calloc(1, sizeof(esp_hdr) + temp_data_len + add_space);
(*hdr)->spi = ntohl(route->spi);
(*hdr)->seq_number = ntohl(route->seq_number++);

int encr_len = route->encr_algo->encrypt(route->encr_data,
  (*hdr)->data, temp_data, temp_data_len);
```

Listing 6.7: Encryption Step of creating a new ESP header

When creating an ESP header, the following obstacles require additional attention:

- The payload of the ESP header is not just meant for user data. For example, *ENCR_AES_CBC_128* uses the first 16 bytes to store its ICV. Thus, the *esp_encr_algo* struct contains a field *additional_space*, specifying how much space of the payload field is required.

- The necessary padding also depends on the utilized encryption algorithm, specified by the *padding* field. While other implementations use the overall padding length as a value for every padding byte, this implementation simply fills them with zeros.
- SPI and Sequence Number have to be converted to the network byte order using *ntohl*.

Due to the header being already allocated and shaped, the processing of received ESP headers is easier and can be broken down into 3 steps:

- First, the SPI is parsed and a linear search for a matching *esp_route* is performed. While the SPI value is theoretically covered by the authentication, it has to be used without validation to actually find the secret state which was associated when the ICV was calculated.
- If a matching *esp_route* has been found, its authentication algorithms verify function is called. Standard authentication algorithms are able to tell whether the data's integrity is still valid, so it either returns *ESP_OK* or *ESP_ERROR*. However, there is also support for delayed authentication algorithms by returning *ESP_MANUAL*, which indicates the decryption step *esp_decrypt_step* will be called manually by the algorithm as soon as authentication is confirmed.
- The last function *esp_decrypt_step* is either called immediately or at a later point in time, depending on the return value of the authentication algorithm. It decrypts the payload according to the *esp_route*, extracts the next header value and calls the callback *on_hdr_processed(route*, next_header, data*, data_len, ...)* on success.

This parsing and creation process is not feature complete, nor does it follow every rule of the original draft. It does not take any IPs into account and does not provide a **sap!** (**sap!**). It allows user-space applications to independently create and process ESP headers with a minimal configuration setup.

6.3.3 Registering Algorithms

```c
typedef struct esp_auth_algo {
  const char* name;
  unsigned int len;
  int (*sign)(void* auth_data, __u8* dest, unsigned char* data,
    unsigned int data_len);
  ESP_RESPONSE (*verify)(void* auth_data, unsigned char* digest,
    unsigned char* data, unsigned int data_len);
} esp_auth_algo;
```

Listing 6.8: ESP Authentication Algorithm Architecture

With the above abstraction, applications can initialize their own *esp_auth_algo* structs, backed by self defined verify and sign functions. They will be added to a static array with a call to the *esp_register_auth_algo(auth_algo*)* function. Every registered algorithm can then be fetched by it's unique name, for example *esp_get_auth_algo("AUTH_NONE")*. The encryption algorithms can be registered in the same manner, however, the encryption algorithm struct contains two more fields for the required padding and a possible IV length. 2 of the 4 pre-existing algorithms have been implemented with the help of OpenSSL's *EVP* API, which is explained in detail in [Opeb].

6.3.4 Test Results

Given the functionality to create and process ESP packets, the question arises whether the library does it in the correct way. To ascertain correct functionality, two different methods can be used:

The first method utilizes another IPsec implementation. By manually establishing bidirectional SAs between a host, which runs this implementation, and another host, which runs another well-tested implementation, one can test whether ESP packets can be successfully exchanged. As a reference implementation, the Linux Kernel's IPsec implementation is a good candidate for such tests.

The second test method utilizes a network protocol analyzer. For this test process, the implementation creates a ESP packet and sends it over the wire. With the help of the analyzer, the packet can be captured and analyzed in detail. An example for such analyzer is Wireshark, which is also widely-used and features support for a lot of different protocols, including the whole IPsec suite. It allows users to manually register SAs and once an IPsec packet is captured, it will use the supplied SAs to verify and decrypt it. Since it also supports all required and even optional algorithms, this can be seen as an IPsec implementation testing as well.

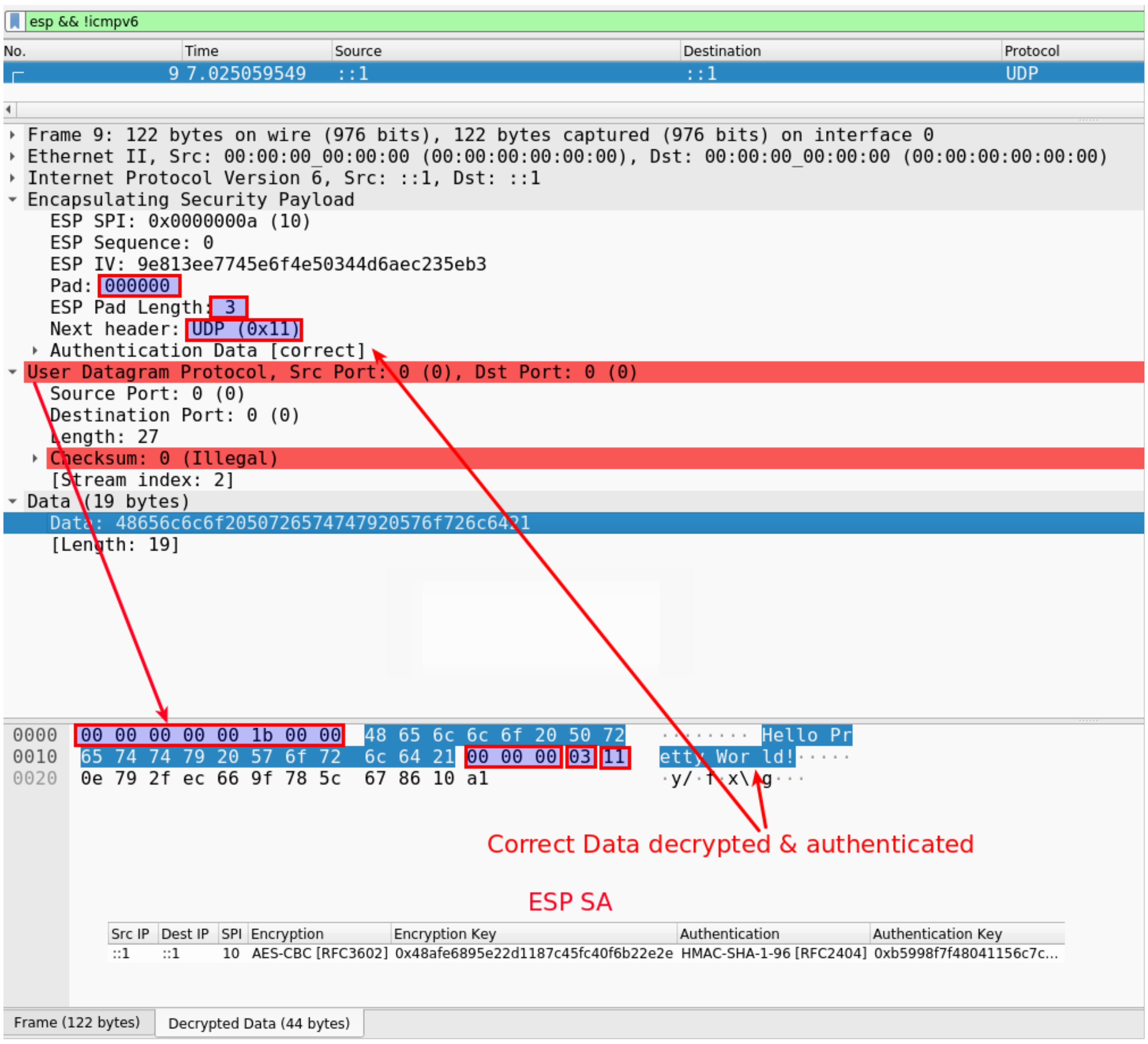
Due to time constraints, just the second test method together with Wireshark is evaluated. A small test application, utilizing the library and Wireshark both share the following SA:

- **Outgoing and Incoming IP**: *::1* (Just Wireshark)
- **SPI**: 10
- **Encryption**: *ENCR_AES_CBC_128* (Key: *0x48afe6895e22d1187c45fc40f6b22e2e*)
- **Authentication**: *AUTH_HMAC_SHA1_96* (Key: *0xb5998f7f48041156c7c344026bf48e63fc28ce2d*)

The test application then constructs a secured ESP packet, containing a sample UDP packet, and sends it to the loopback interface, with *::1* being the destination and source address (A more detailed explanation on how to send ESP packets is given in 6.4.2). Figure 6.4 shows how Wireshark is able to successfully interpret, authenticate and decrypt the captured packet, which implies that the packet was correctly formed according to the specification. While this theoretically just proves packets are correctly created, the test application in chapter 6.4 will also process these packets, showing this is correctly working as well.

6.4 Test Application

The Test Application combines both libraries to create a prototype, TESLA based ESP IPv6 multicast stream between a single sender (server) and multiple receivers (clients). In detail, the two phases of the design chapter are implemented the following way:
As soon as the sender application starts, it sets up a TESLA instance. While some parameters, like the utilized algorithm can be configured with environment variables, some are predefined. After that, a 30 second long bootstrap phase starts. In this phase, every interested receiver can send a ESP protected *client_bootstrap_request_msg* to the sender. The ESP packet is encrypted and authenticated by a unique pair of PSKs and identified by a unique SPI, shared between the server and a single client. Thus, the server has to be



```
esp && !icmpv6
No.        Time            Source          Destination     Protocol
      9 7.025059549        ::1             ::1             UDP

▸ Frame 9: 122 bytes on wire (976 bits), 122 bytes captured (976 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 6, Src: ::1, Dst: ::1
▾ Encapsulating Security Payload
    ESP SPI: 0x0000000a (10)
    ESP Sequence: 0
    ESP IV: 9e813ee7745e6f4e50344d6aec235eb3
    Pad: 000000
    ESP Pad Length: 3
    Next header: UDP (0x11)
  ▸ Authentication Data [correct]
▾ User Datagram Protocol, Src Port: 0 (0), Dst Port: 0 (0)
    Source Port: 0 (0)
    Destination Port: 0 (0)
    Length: 27
  ▸ Checksum: 0 (Illegal)
    [Stream index: 2]
▾ Data (19 bytes)
    Data: 48656c6c6f20507265747479205776f726c6421
    [Length: 19]

0000  00 00 00 00 00 1b 00 00  48 65 6c 6c 6f 20 50 72   ········Hello Pr
0010  65 74 74 79 20 57 6f 72  6c 64 21 00 00 00 03 11   etty Wor ld!·····
0020  0e 79 2f ec 66 9f 78 5c  67 86 10 a1               ·y/·f·x\ g···

Frame (122 bytes)   Decrypted Data (44 bytes)
```

Src IP	Dest IP	SPI	Encryption	Encryption Key	Authentication	Authentication Key
::1	::1	10	AES-CBC [RFC3602]	0x48afe6895e22d1187c45fc40f6b22e2e	HMAC-SHA-1-96 [RFC2404]	0xb5998f7f48041156c7c...

Figure 6.4: Wireshark deconstructing the ESP packet

configured with the amount of possible receivers to correctly setup the required SAs. For the sake of simplicity, the prototype's bootstrap strategy does not involve any IKEv2 or G-IKEv2 operations and only offers a one time bootstrap to the clients. Once it receives the above mentioned bootstrap request, it answers with a similar encrypted and authenticated *server_bootstrap_response_msg* ESP packet, providing the client with all TESLA parameters of the upcoming stream. After the 30 seconds elapsed, the data stream phase begins.
In this phase, the server starts sending out TESLA authenticated ESP packets. As payload for each packet, we define the following test data: The server reads and process a file called *words.txt*. It splits it into an array, each entry containing a line of this text file. For each line, it then emulates a data stream by broadcasting the line contents as the payload of a ESP packet. Instead of sending all packets in one TESLA interval, we will send one packet in each interval. Thus, the hash chain at the sender must be at least the amount of lines in the file. Additionally, to allow authenticating the last d packets, where d is the disclosure

delay, the sender's hash chain length n is calculated using this formula:

$$n = \text{line\_count}(\text{"words.txt"}) + d \tag{6.1}$$

The last d packets simply include the payload "NULL". On the other side, the receivers listen for the broadcasted TESLA packets and authenticate them using the previously received TESLA parameters. For test purposes, they simply print the line contents upon authentication.
Since the Test Application primarily consists of configuration and message parsing, it won't be presented as detailed as the libraries. An introduction into the important TESLA ESP extension and IPv6 module is given below.

6.4.1 TESLA ESP Extension

```
int tesla_sign(void* auth_data, __u8* dest, unsigned char* data,
  unsigned int data_len) {
  tesla_core_state* state = (tesla_core_state*) auth_data;
  tesla_auth_header* header;
  tesla_core_get_auth_header(state, data, data_len, &header);
  int len = tesla_auth_header_to_icv(header, dest);
  tesla_auth_header_free(header);
  return len;
}
ESP_RESPONSE tesla_verify(void* auth_data, unsigned char* digest,
  unsigned char* data, unsigned int data_len) {
  tesla_core_state* state = (tesla_core_state*) auth_data;
  tesla_core_verify(state, data, data_len, digest);
  return ESP_MANUAL; // TESLA will call next step
}

// 44 bytes - 4 bytes interval | 20 bytes MAC | 20 bytes Disclosed Key
esp_auth_algo tesla_sha_44 = { "AUTH_TESLA_44", 44, tesla_sign, tesla_verify };
// 68 bytes - 4 bytes interval | 32 bytes MAC | 32 bytes Disclosed Key
esp_auth_algo tesla_sha_68 = { "AUTH_TESLA_68", 68, tesla_sign, tesla_verify };
```

At this point, the TESLA and ESP library are decoupled and TESLA authenticated ESP packets are not generatable yet. The Test Application solves this problem by registering two new ESP authentication algorithms: *AUTH_TESLA_44* (44 bytes ICV) and *AUTH_TESLA_68* (68 bytes ICV). While they are different regarding signature size, they both use the same verify and sign functions. This can be achieved due to *auth_data* being a pointer to a full TESLA instance, knowing the size of the MAC and disclosed key.

6.4.2 Sending IPv6 packets

An important part of the Test Application is receiving and sending ESP packets in the bootstrap and streaming phase. On Linux, sending network packets can be achieved using various calls to the socket API, which is also defined by the POSIX standard. In the following paragraph, an overview how user space applications can send and receive custom IPv6 packets is given:

Sending Similar to normal TCP und UDP sockets, an application can create a so called *RAW_SOCKET* [Mic]:

```
int sock = socket(AF_INET6, SOCK_RAW, IPPROTO_ESP);
```

Listing 6.9: Opening a raw socket with protocol number 50 (IPPROTO_ESP)

This type of socket allows applications to send arbitrary payloads (e.g ESP or AH headers), wrapped by an automatically generated IP header. The last parameter hereby specifies the protocol number, according to the internet protocol number list by the IANA [Int]. Once this socket is opened and an ESP is generated by the library, a call to the *sendto* function sends it:

```
struct sockaddr_in6 dest;
dest.sin6_family = AF_INET6;
inet_pton(AF_INET6, "::1", &(dest.sin6_addr));

esp_hdr* hdr;                              //253 = reserved next header
int hdr_len = esp_hdr_init(route, &hdr, 253, data, data_len);
sendto(sock, hdr, hdr_len, 0, &dest, sizeof(struct sockaddr_in6));
```

Listing 6.10: Sending ESP Headers to address ::1

Hereby, it uses the common *sockaddr_in6* struct to specify the destination IP. The source IP is filled in automatically, which is not suited for node local tests, since all packets will have the same source IP. Instead, we utilize the *bind* function to set it manually:

```
struct sockaddr_in6 src;
src.sin6_family = AF_INET6;
inet_pton(AF_INET6, "::2", &(src.sin6_addr))
bind(sock, &src, sizeof(struct sockaddr_in6)
```

Listing 6.11: Setting the source IP via *bind*

All outgoing packets are routed to the default network interface, which often is either *eth0* for cable connections or *wlp1s0* for wireless connections. To change this behavior, it can either be changed on the Linux platform by using the *ip* tool or directly on the socket by setting the *SO_BINDTODEVICE* option to an interface name. Outgoing packets directed to an multicast address even need to have the socket option *IPV6_MULTICAST_IF* set correctly.

```
// Linux Platform level - IP CLI Tool
ip -6 route add ff01::1 dev lo
// or on socket Level, where iface is "eth0" or similar
setsockopt(socket, SOL_SOCKET, SO_BINDTODEVICE, iface, sizeof(iface));
// Multicast addresses only: ifindex is the interface index of e.g "eth0"
setsockopt(socket, IPPROTO_IPV6, IPV6_MULTICAST_IF, &ifindex, sizeof(ifindex))
```

Receiving The logic to receive IPv6 packets follows the same principle:

```
int socket = socket(AF_INET6, SOCK_RAW, IPPROTO_ESP);
int esp_len = recvfrom(socket, buffer, buffer_len, 0, src_addr, src_addr_len);
```

Listing 6.12: Opening a raw socket and receive a packet

This socket now will receive all ESP packets. However, especially in the bootstrap phase, clients just want to receive packets from the server IP: A similar call to *bind* is required, which constrains received packets to a specific destination IP. Unfortunately, this does not seem to work for IPv6 multicast IPs. Generally speaking, receiving packets via multicast IPs requires the socket to join the multicast group first:

```
struct ipv6_mreq command;
command.ipv6mr_interface = ifindex; // Network Interface index
memcpy(&command.ipv6mr_multiaddr, &addr->sin6_addr, sizeof(struct in6_addr));
setsockopt(socket, IPPROTO_IPV6, IPV6_ADD_MEMBERSHIP, &command, sizeof(command));
```

Listing 6.13: Join an IPv6 multicast group

The Test Applications wraps this logic into a module **ipv6.{c | h}**, providing IPv6 receive and send functions. It also contains additional logic, for example keeping the socket state and setting specific timeouts for receive calls, which are not explained in more detail.

6.5 Evaluation

With the above mentioned implementations in place, this chapter focuses on the evaluation of it. It starts with testing the Test Application on the FIT IoT-LAB[IL18b] platform. In different scenarios, multiple A8 nodes (600 Mhz CPU and 256 MB RAM) are deployed and setup to test the unicast-based bootstrap and the multicast-based TESLA data stream. Following up, the hash chain performance is analyzed on A8 nodes regarding it's overall resource usage, including the creation time and required storage. The chapter is concluded with an overview of possible different implementation strategies and opportunities.

6.5.1 IoT-Lab Test

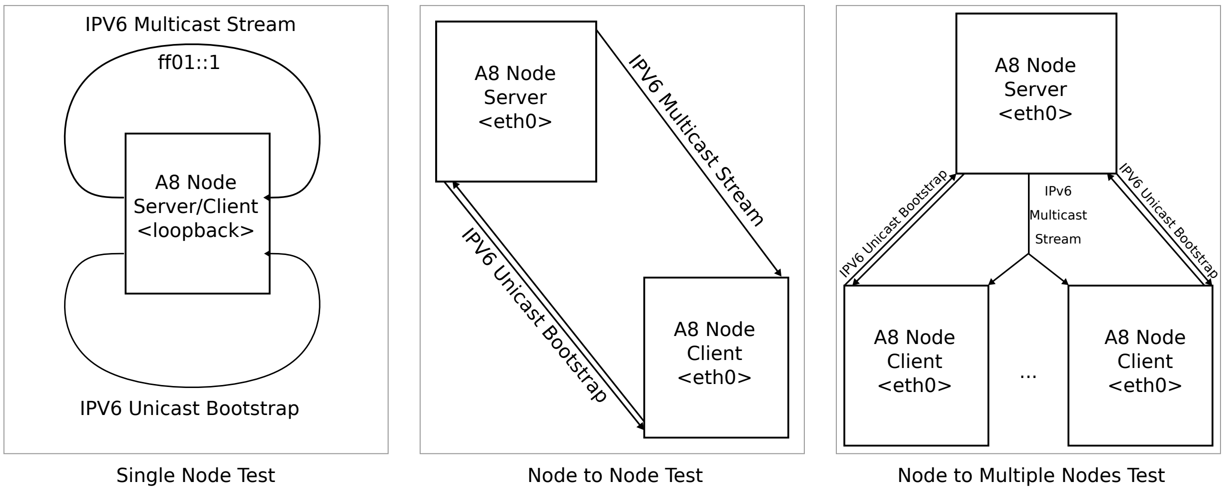


Figure 6.5: Iot-LAB Test Setup

To verify that the Test Application, including the utilized libraries, work correctly, the FIT Iot-LAB is used to create multiple different test scenarios. All test scenarios use A8 nodes, which are based on the ARM architecture and feature a high-performance ARM Cortex-A8 microprocessor, running at a clock speed of 600 Mhz, and 256 MB of RAM [IL18a]. In general, these nodes are very powerful in comparison to some devices utilized in the IoT world. They are even able to run a full sized Linux distribution. Thus, in a real world example they would be seen as larger control nodes. In the Phillips Hue example of chapter 1.1, such node is a perfect candidate for a bridge controller, whereas the actual lights may run on a light platform such as RIOT OS. If not mentioned otherwise, the following TESLA parameters will be chosen for the tests:

- **Interval Duration: 200ms**
- **Disclosure Delay: 4**
- **Algorithm: SHA256 (Message, Message Key and Hash Chain)**
- **Hash Chain length: Defined by amount of lines in *words.txt***

The test scenarios will consist of a single node test, a single node to single node test and a larger single node to multiple nodes test, as shown in Figure 6.5.

Single Node Test The first test tries to eliminate any external routing or network problems by running the sender and client application on the same node. Both applications are configured to use the *loopback* interface, which makes sure the traffic does not leave the node. Additionally, a link-local IPv6 multicast address and a localhost IPv6 address are used for the bootstrap and data stream phase.
After the single node experiment started and the projects were compiled for the ARM platform, the following setup has to be executed on the node:

```
ifconfig lo multicast # Enable multicast for Loopback
ip -6 addr add ::2 dev lo # Add another IPv6 address to Loopback
ip -6 route add ff01::1 dev lo # Add route to LO for IPv6 multicast address
```

Listing 6.14: Linux Loopback Setup Process

While the first simply enables multicast support on the loopback interface, the next two lines add a new address and route. The address is needed to identify and make the requesting client addressable, since the server already uses the localhost IP *::1*. The last line adds a special route for the address *ff01::1*, leading to the loopback device (Surprisingly, this change was necessary, although the underlying socket already specified *lo* as the outgoing/incoming interface. Without this line, the client was not able to receive packets). After the setup, both binaries are run on the same node:

```
IFACE=lo IP=::1 BROADCAST_IP=ff01::1 ./tesla_server
IFACE=lo IP=::2 BROADCAST_IP=ff01::1 SEVER_IP=::1 ./tesla_client
```

Listing 6.15: Test Application Runtime Configuration

The end results are very positive: The client could correctly be bootstraped by the server via unicast messages. In the data streaming phase, the client was able to receive and authenticate all packets, printing the whole content of the *words.txt* file.

```
# server - First packet contains the line "Hello":
15:01:06 INFO  src/server/main.c:105: [1541685666756] Sending packet 0
# client - First packet authenticated:
```

```
15:01:07 INFO  src/client/main.c:29:  [1541685667589] Authenticated Line: Hello
```

Listing 6.16: Application Log Syllabus

The above log syllabus shows, that the client authenticated the message at a correct point in time: The server sent the first packet at $T_s = 1541685666.756$ and the client authenticated it at $T_r = 1541685667.589$. The earliest point in time on which the packet would be authenticable, is $T_{auth} = T_s + (200 * 4) = 1541685667.556$, ignoring processing and transfer delays. Due to $T_r \geq T_{auth}$, a correct authentication process can be assumed.

Node to Node Test The next test happens between two A8 nodes. One is selected as the server node and the other one is the client node. For this test, no setup is required: Every A8 node has its own IP, so no additional IP has to be added to an interface. Additionally, multicast traffic is already routed to *eth0* by default, so no route has to be added. The IPv6 multicast address is changed to *ff0e::1*, which is listed in the global address list and allows the routing/forwarding of packets outside of the node/link [ian18]. After inspecting the global IPs of both nodes, the applications are started with the following configuration:

```
IFACE=eth0 IP=2001:660:5307:3000::67 BROADCAST_IP=ff0e::1 ./tesla_server
IFACE=eth0 IP=2001:660:5307:3000::68 BROADCAST_IP=ff0e::1 \
  SERVER_IP=2001:660:5307:3000::67 ./tesla_client
```

Listing 6.17: Test Application Runtime Configuration

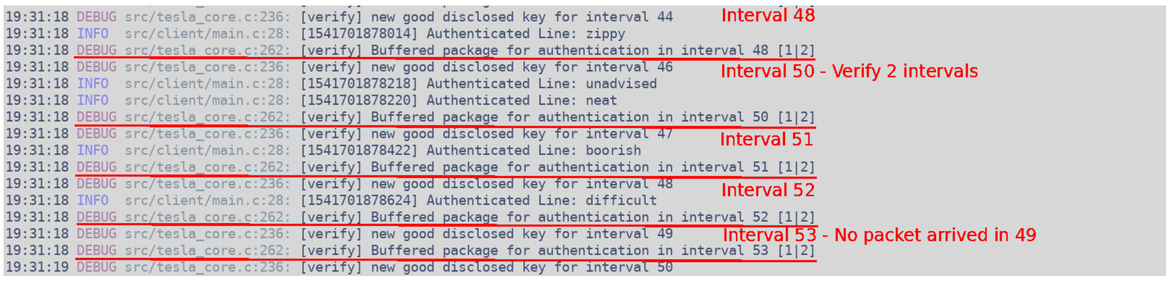
The results are equally positive. The bootstrap and data stream were successfull. A small difference to the previous test result was the following: The previous link-local bootstrap yielded a clock lag of 16ms for the client (While theoretically, the clock lag should be 0, it includes the duration of the bootstrap packet's sending, encryption and verifying process). In this test, the bootstrap resulted in a clock drift of 24ms. This difference is due to the small network delay between the nodes, which are already close to each other location-wise. However, this had no impact on the authentication process and is expected.

Single Node to 7 Nodes Test The last test ensures the functionality of the IPv6 multicast broadcast by running 7 clients in total. Furthermore, the test data size is increased: The previous *words.txt* contained 12 lines with a single word per line. This amount is increased to 100 lines with one word per line, resulting in a longer hash chain and transmission duration.

```
IFACE=eth0 IP=2001:660:5307:3000::67 BROADCAST_IP=ff0e::1 ./tesla_server
# On 7 different clients:
IFACE=eth0 IP=<node_ip> BROADCAST_IP=ff0e::1 \
  SERVER_IP=2001:660:5307:3000::67 ./tesla_client
```

Listing 6.18: Test Application Runtime Configuration

The test was also successful and every client authenticated all packets. Due to the longer transmission, the following phenom occured: To emulate processing time, the Test Application server simply sleeps T_{int} milliseconds after sending a packet, where T_{int} is the interval duration. Due to additional timing overhead, this simple timer was not aligned correctly anymore, which made the server skip an interval. Thus, the client did not receive a disclosed key for a previous interval. However, with the help of a later disclosed key, it was able to recalculate the lost disclosed key and authenticate as usual (Figure 6.6).



```
19:31:18 DEBUG src/tesla_core.c:236: [verify] new good disclosed key for interval 44
19:31:18 INFO  src/client/main.c:28: [1541701878014] Authenticated Line: zippy
19:31:18 DEBUG src/tesla_core.c:262: [verify] Buffered package for authentication in interval 48 [1|2]
19:31:18 DEBUG src/tesla_core.c:236: [verify] new good disclosed key for interval 46
19:31:18 INFO  src/client/main.c:28: [1541701878218] Authenticated Line: unadvised
19:31:18 INFO  src/client/main.c:28: [1541701878220] Authenticated Line: neat
19:31:18 DEBUG src/tesla_core.c:262: [verify] Buffered package for authentication in interval 50 [1|2]
19:31:18 DEBUG src/tesla_core.c:236: [verify] new good disclosed key for interval 47
19:31:18 INFO  src/client/main.c:28: [1541701878422] Authenticated Line: boorish
19:31:18 DEBUG src/tesla_core.c:262: [verify] Buffered package for authentication in interval 51 [1|2]
19:31:18 DEBUG src/tesla_core.c:236: [verify] new good disclosed key for interval 48
19:31:18 INFO  src/client/main.c:28: [1541701878624] Authenticated Line: difficult
19:31:18 DEBUG src/tesla_core.c:262: [verify] Buffered package for authentication in interval 52 [1|2]
19:31:18 DEBUG src/tesla_core.c:236: [verify] new good disclosed key for interval 49
19:31:18 DEBUG src/tesla_core.c:262: [verify] Buffered package for authentication in interval 53 [1|2]
19:31:19 DEBUG src/tesla_core.c:236: [verify] new good disclosed key for interval 50
```

Interval 48
Interval 50 - Verify 2 intervals
Interval 51
Interval 52
Interval 53 - No packet arrived in 49

Figure 6.6: Client Log File

With the above tests, the general functionality of both libraries has been successfully tested and verified. As an additional safety test, the tool Valgrind was used to determine if any memory leaks occur when running the application [val18]. Valgrind provides memory leak detection by hooking into various functions of the standard library and is able to determine when a pointer to some allocated space is lost or never freed. However, the usage of Valgrind on an A8 node was not possible: Valgrind significantly slows down the code due its hooking and analyzing system. Thus, the client application was not able to receive all the sent packets. To circumvent this, the Valgrind test was run on a more powerful, conventional laptop, where the slow down was not as significant. The test yielded no possible memory leaks.

6.5.2 RIOT Implementation Test

As an additon, a cross-platform test between this and Christopher Schütze's implemenation [Sch] has been accomplished. The test occured on a single Linux machine, on which the Linux implementation was started as the server and the RIOT implementation was started as a native client. The communication between both processes was established with the help of a TUN/TAP virtual network interface.
The bootstrap phase required some changes on behalf of both. While Schütze's implementation had a seperate two-way message exchange for the time synchronization, the Linux one included this process in the two-way TESLA bootstrap exchange, as seen in Figure 5.2. Additionally, the Linux version used an, generally optional, ESP encryption for the initital bootstrap, which has been removed due to incompatability. After the bootstrap message structs and the PSKs had been synchronized and exchanged, the bootstrap was successful. The RIOT client was able to receive and parse all important parameters. Especially, the time synchronization presented in chapter 5.2 was a success: While the Linux TESLA implementation uses absolute unix timestamps for general timekeeping, RIOT uses elapsed milliseconds since initital boot. Nevertheless, the client was able to correctly calculate an upper-bound of the sender time.
After the initialization, the attempt to authenticate received packets at the client was not successful. Both libraries implemented the TESLA ICV differently, resulting in incompatabile ESP packets. Due to time constraints, further test runs with compatible headers can not be documented as part of this thesis.

6.5.3 Hash Chain Performance

On behalf of the sender, the creation and calculation of the hash chain can be considered as one of the most expensive operations. Afterwards, just 2 cryptographic operations per outgoing packet are required. To grasp the possibilities of the A8, 5 different hash algorithms are tested together with 5 different hash chain lengths on a node. A special test program, which benchmarks the allocation and filling process of a hash chain, is deployed to the node and is run 3 times per algorithm/hash-chain combination. The test results include the average creation duration (from memory allocation to a calculcated hash chain, not including memory freeing), required memory and the theoretical TESLA ICV length (In the case of the algorithm being used as a hash chain and message HMAC algorithm). The results are presented in 6.1. For smaller chain lengths e.g 100,000, the algorithm choice seems to have a rather unessential influence. Thus, if the network and receivers support slighly larger ICVs, the usage of SHA256 should be preferred over SHA1 for security reasons mentioned in 2.1. Depending on the RAM utilization, an A8 node is able to store a one million long SHA256 hash chain (32 MB) and additionally generate and store a second one for the next transmission in under 4 seconds. With an interval duration of 200ms, this would allow a total transmission duration of 2 days, 7 hours and 33 minutes with a single bootstrap.
The combination of SHA384/SHA512 with a hash chain length of 5 million exceeded the RAM capabilities of the A8 node and resulted in a program crash.
The overall results confirm, that an A8 node is a good candiate for a TESLA-based bridge controller.

6.5.4 Alternative Implementation Strategies

For this prototype, the implementation was split into two user-space libraries and a user-space application. However, there are some alternative implementation possibilities on Linux, which are explained below: Besides implementing the TESLA protocol as a standalone user-space library, a kernel module providing a TESLA *Crypto-API* implementation was also considered. When looking at different authentication algorithms, like HMAC, there is quite a similarity regarding its operations to the TESLA ones. The first operation is signing: Given an input and a state, an ICV can be calculated. In most of the algorithms, the state is a secret key with a specific length. The second operation is verifying an input with its ICV and the same state used to generate the ICV. While it seems like the operations are nearly identical, the following aspects made an implementation as *Crypto-API* authentication algorithm impractical:

A more complex State: In comparison to a simple secret key, the TESLA state is more complex. It contains multiple different values of different types, including a full size hash chain for senders. Additionally, packets have to be cached at the receiver and accessed at a later point in time, making the state bigger, mutable over time and hard to manage.

Delayed Verification required: The *Crypto-API* provides a well thought out Application Programming Interface (API) to register new authentication algorithms. However, it is unsuitable for TESLA because it requires an immediate verification result, which TESLA is unable to provide. Thus, a larger internal API change would be necessary to add TESLA support to the *Crypto-API*.

Another consideration was the implementation of a TESLA *strongSwan* plugin [str]. However, it would meet the same missing delayed authentication API problem as men-

Algorithm	**Hash Size (TESLA ICV Size)**	**Chain Length**	**$\tilde{T}$ime in *ms***	**Size**
SHA1	20B (44B)	100	7	2 *kB*
		1,000	20	20 *kB*
		100,000	465	2 *MB*
		1,000,000	2,836	20 *MB*
		5,000,000	13,435	100 *MB*
SHA224	28B (60B)	100	10	2.8 *kB*
		1,000	22	28 *kB*
		100,000	537	2.8 *MB*
		1,000,000	3,620	28 *MB*
		5,000,000	17,335	140 *MB*
SHA256	32B (68B)	100	10	3.2 *kB*
		1,000	24	32 *kB*
		100,000	539	3.2 *MB*
		1,000,000	3,722	32 *MB*
		5,000,000	18,045	160 *MB*
SHA384	48B (100B)	100	11	4.8 *kB*
		1,000	43	48 *kB*
		100,000	947	4.8 *MB*
		1,000,000	7,788	48 *MB*
		5,000,000	—	240 *MB*
SHA512	64B (132B)	100	11	6.4 *kB*
		1,000	43	64 *kB*
		100,000	983	6.4 *MB*
		1,000,000	8,001	64 *MB*
		5,000,000	—	320 *MB*

Table 6.1: Hash Chain Creation Test Results on an A8 Node

tioned with the *Crypto-API*. An advantage of a strongSwan plugin would be the support of G-IKEv2 [Eng].

7 Conclusion and Future Work

With the last two chapters, we successfully presented and evaluated a Linux based TESLA implementation. While it is still a prototype, its design as a library allows other, future implementations to make use of it. In addition and to support the use of TESLA ICVs in the ESP protocol, a minimal ESP library was developed, allowing applications to integrate custom encryption and authentication algorithms. The prototype Test Application connected both libraries and with the help of raw sockets, a successful bootstrap and TESLA data stream could be accomplished. Both libraries still have to be treated as prototypes and should not be considered feature complete or bug free. In detail, the ESP library currently does not have any kind of traffic selector nor SPD support. The memory usage of the TESLA library has not been optimized yet: A TESLA receiver currently has to allocate a hash chain of the same length as the sender's one, although just a single key would suffice. In addition to the TESLA data stream as such, two different bootstrap designs, including the possible use of IKEv2 and G-IKEv2, were proposed. Thus, the thesis did not only combine TESLA ICVs with ESP, but also presented a possible TESLA IPsec integration. Existing IPsec and cryptographic suites unfortunately did not have the required extensibility for the addition of a delayed authentication protocol. A larger API change was impractical for this thesis, whereupon the standalone prototypes were the preferred approach.
Summarizing, the thesis successfully showed the integration of the TESLA protocol into the ESP stack on the Linux platform.

TESLA itself turns out to be a very flexible and adaptable authentication protocol. The parameters allow users and applications to precisely calibrate TESLA to the underlying scenario. However, one of its biggest disadvantages does not reside in the protocol itself but rather in a requirement: The need of an initial and a subsequent bootstrap of TESLA parameters. While this may not be a problem for large senders as seen in chapter 6.5.3, which can broadcast for multiple days without the need for a re-key process, smaller senders would suffer from frequently needed re-keying processes. Similar, the required receiver buffering can also be seen as a rather large disadvantage. Thus, a possible future work could consist of combining the best properties of some TESLA derivatives, like TESLA++ and inf-TESLA, into a single one. In this example, the combination of TESLA++ and inf-TESLA would feature a smaller required receiver buffer, an endless TESLA stream and a better DoS protection with the drawback of a larger message size and a higher memory footprint at the sender.
Additionally, future work may deal with a complete TESLA-based smart home scenario, similar to the Phillips Hue example mentioned in chapter 1.1. This setup then could be used to analyze and compare different multicast authentication solutions, including the related work presented in chapter 4.
While the current IPsec and cryptographic backends do not support delayed authentication protocols yet, it can be considered as future work. This would allow the integration of this library and possible other protocols, like EMSS and TESLA derivatives.

List of Figures

Bibliography

[BLMQ05] Barreto, Paulo S. L. M. ; Libert, Benoît ; McCullagh, Noel ; Quisquater, Jean-Jacques: Efficient and Provably-Secure Identity-Based Signatures and Signcryption from Bilinear Maps. In: Roy, Bimal (Hrsg.): *Advances in Cryptology - ASIACRYPT 2005*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2005. – ISBN 978–3–540–32267–2, S. 515–532

[CAPC16] Câmara, Sérgio ; Anand, Dhananjay ; Pillitteri, Victoria ; Carmo, Luiz: Multicast Delayed Authentication For Streaming Synchrophasor Data in the Smart Grid. In: *IFIP International Information Security and Privacy Conference* Springer, 2016, S. 32–46

[Cry] Crypto++ community (Hrsg.): *Crypto++ 6.0.0 Benchmarks*. Speed Comparison of Popular Crypto Algorithms: Crypto++ community, `https://www.cryptopp.com/benchmarks.html`

[DR08] Dierks, T. ; Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2 / RFC Editor. Version: August 2008. `http://www.rfc-editor.org/rfc/rfc5246.txt`. RFC Editor, August 2008 (5246). – RFC. – ISSN 2070–1721. – `http://www.rfc-editor.org/rfc/rfc5246.txt`

[Ekl] Eklektix, Inc. (Hrsg.): *Kernel development*. lwn: Eklektix, Inc., `https://lwn.net/Articles/13587/`

[Eng] Engelbrecht, Wolfgang: *Group Key Management with Strongswan*

[Hol18] Holding, Signify: *Philips Hue Smart Home Lampen — Philips Hue*. `https://www2.meethue.com/de-de`. Version: 2018. – Accessed on 24.10.2018

[ian18] Internet Assigned Numbers Authority: *IPv6 Multicast Address Space Registry*. `https://www.iana.org/assignments/ipv6-multicast-addresses/ipv6-multicast-addresses.xhtml`. Version: 2018. – Accessed on 08.11.2018

[IL18a] IoT-Lab: *A8 open node - FIT/IoT-LAB*. `https://www.iot-lab.info/hardware/a8/`. Version: 2018. – Accessed on 08.11.2018

[IL18b] IoT-Lab: *IoT-LAB: a very large scale open testbed*. `https://www.iot-lab.info/`. Version: 2018. – Accessed on 24.08.2018

[Int] Internet Assigned Numbers Authority (Hrsg.): *Protocol Numbers*. IANA Assignments: Internet Assigned Numbers Authority, `https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml`

[JK03] Jonsson, J. ; Kaliski, B.: Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1 / RFC Editor. Version: February 2003. http://www.rfc-editor.org/rfc/rfc3447.txt. RFC Editor, February 2003 (3447). – RFC. – ISSN 2070–1721. – http://www.rfc-editor.org/rfc/rfc3447.txt

[KBC97] Krawczyk, Hugo ; Bellare, Mihir ; Canetti, Ran: HMAC: Keyed-Hashing for Message Authentication / RFC Editor. Version: February 1997. http://www.rfc-editor.org/rfc/rfc2104.txt. RFC Editor, February 1997 (2104). – RFC. – ISSN 2070–1721. – http://www.rfc-editor.org/rfc/rfc2104.txt

[Ken05a] Kent, S.: IP Authentication Header / RFC Editor. Version: December 2005. http://www.rfc-editor.org/rfc/rfc4302.txt. RFC Editor, December 2005 (4302). – RFC. – ISSN 2070–1721. – http://www.rfc-editor.org/rfc/rfc4302.txt

[Ken05b] Kent, S.: IP Encapsulating Security Payload (ESP) / RFC Editor. Version: December 2005. http://www.rfc-editor.org/rfc/rfc4303.txt. RFC Editor, December 2005 (4303). – RFC. – ISSN 2070–1721. – http://www.rfc-editor.org/rfc/rfc4303.txt

[KHN+14] Kaufman, C. ; Hoffman, P. ; Nir, Y. ; Eronen, P. ; Kivinen, T.: Internet Key Exchange Protocol Version 2 (IKEv2) / RFC Editor. Version: October 2014. http://www.rfc-editor.org/rfc/rfc7296.txt. RFC Editor, October 2014 (79). – STD. – ISSN 2070–1721. – http://www.rfc-editor.org/rfc/rfc7296.txt

[KS05] Kent, S. ; Seo, K.: Security Architecture for the Internet Protocol / RFC Editor. Version: December 2005. http://www.rfc-editor.org/rfc/rfc4301.txt. RFC Editor, December 2005 (4301). – RFC. – ISSN 2070–1721. – http://www.rfc-editor.org/rfc/rfc4301.txt

[Lam81] Lamport, Leslie: Password Authentication with Insecure Communication. In: *Commun. ACM* 24 (1981), November, Nr. 11, 770–772. http://dx.doi.org/10.1145/358790.358797. – DOI 10.1145/358790.358797. – ISSN 0001–0782

[lin] IoT Developer Survey 2016. https://www.slideshare.net/IanSkerrett/iot-developer-survey-2016

[Mel] Melnikov, Andrian: *A Testbed for Evaluating ID-based Authentication in Constrained Networks*

[Mic] Michael Kerrisk (Hrsg.): *RAW(7) Linux Manual Page*. Linux Manual Page: Michael Kerrisk, http://man7.org/linux/man-pages/man7/raw.7.html

[MMPR11] M'Raihi, D. ; Machani, S. ; Pei, M. ; Rydell, J.: TOTP: Time-Based One-Time Password Algorithm / RFC Editor. Version: May 2011. http://www.rfc-editor.org/rfc/rfc6238.txt. RFC Editor, May 2011 (6238). – RFC. – ISSN 2070–1721. – http://www.rfc-editor.org/rfc/rfc6238.txt

[Opea] OpenSSL Foundation, Inc (Hrsg.): */docs/manmaster/man1/enc.html*. OpenSSL Docs: OpenSSL Foundation, Inc, `https://www.openssl.org/docs/manmaster/man1/enc.html`

[Opeb] OpenSSL Foundation, Inc (Hrsg.): *EVP*. OpenSSL Wiki: OpenSSL Foundation, Inc, `https://wiki.openssl.org/index.php/EVP`

[Opec] OpenSSL Foundation, Inc (Hrsg.): *OpenSSL*. OpenSSL: OpenSSL Foundation, Inc, `https://www.openssl.org/`

[PCTS00] Perrig, A. ; Canetti, R. ; Tygar, J. D. ; Song, Dawn: Efficient authentication and signing of multicast streams over lossy channels. In: *Proceeding 2000 IEEE Symposium on Security and Privacy. S P 2000*, 2000. – ISSN 1081–6011, S. 56–73

[PP18] Park, C. ; Park, W.: A Group-Oriented DTLS Handshake for Secure IoT Applications. In: *IEEE Transactions on Automation Science and Engineering* 15 (2018), Oct, Nr. 4, S. 1920–1929. `http://dx.doi.org/10.1109/TASE.2018.2855640`. – DOI 10.1109/TASE.2018.2855640. – ISSN 1545–5955

[PSC+05] Perrig, A. ; Song, D. ; Canetti, R. ; Tygar, J. D. ; Briscoe, B.: Timed Efficient Stream Loss-Tolerant Authentication (TESLA): Multicast Source Authentication Transform Introduction / RFC Editor. RFC Editor, June 2005 (4082). – RFC. – ISSN 2070–1721

[PST+02] Perrig, Adrian ; Szewczyk, Robert ; Tygar, J.D. ; Wen, Victor ; Culler, David E.: SPINS: Security Protocols for Sensor Networks. In: *Wireless Networks* 8 (2002), Sep, Nr. 5, 521–534. `http://dx.doi.org/10.1023/A:1016598314198`. – DOI 10.1023/A:1016598314198. – ISSN 1572–8196

[RSA78] Rivest, Ronald L. ; Shamir, Adi ; Adleman, Leonard: A method for obtaining digital signatures and public-key cryptosystems. In: *Communications of the ACM* 21 (1978), Nr. 2, S. 120–126

[SBBP09] Studer, A. ; Bai, F. ; Bellur, B. ; Perrig, A.: Flexible, extensible, and efficient VANET authentication. In: *Journal of Communications and Networks* 11 (2009), Dec, Nr. 6, S. 574–588. `http://dx.doi.org/10.1109/JCN.2009.6388411`. – DOI 10.1109/JCN.2009.6388411. – ISSN 1229–2370

[SBK+17] Stevens, Marc ; Bursztein, Elie ; Karpman, Pierre ; Albertini, Ange ; Markov, Yarik: The first collision for full SHA-1. In: *Annual International Cryptology Conference* Springer, 2017, S. 570–596

[Sch] Schütze, Christopher: *RIOT TESLA*

[Sha85] Shamir, Adi: Identity-Based Cryptosystems and Signature Schemes. In: Blakley, George R. (Hrsg.) ; Chaum, David (Hrsg.): *Advances in Cryptology*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1985. – ISBN 978–3–540–39568–3, S. 47–53

[Ste06] Stevens, Marc: Fast Collision Attack on MD5. In: *IACR Cryptology ePrint Archive* 2006 (2006), S. 104

[str] STRONGSWAN (Hrsg.): *strongSwan - IPsec VPN for Linux, Android, FreeBSD, Mac OS X, Windows.* strongSwan Homepage: strongSwan, `https://www.strongswan.org/`

[TK05] THOMSEN, Søren S. ; KNUDSEN, Lars R.: *Cryptographic hash functions*, PhD thesis, Technical University of Denmark, Diss., 2005

[TNR17] TILOCA, Marco ; NIKITIN, Kirill ; RAZA, Shahid: Axiom: Dtls-based secure iot group communication. In: *ACM Transactions on Embedded Computing Systems (TECS)* 16 (2017), Nr. 3, S. 66

[val18] *Valgrind Home*. `http://www.valgrind.org/`. Version: 2018. – Accessed on 08.11.2018

[WMM+17] WOUTERS, P. ; MIGAULT, D. ; MATTSSON, J. ; NIR, Y. ; KIVINEN, T.: Cryptographic Algorithm Implementation Requirements and Usage Guidance for Encapsulating Security Payload (ESP) and Authentication Header (AH) / RFC Editor. RFC Editor, October 2017 (8221). – RFC. – ISSN 2070–1721

[WS18] WEIS, Brian ; SMYSLOV, Valery: Group Key Management using IKEv2 / IETF Secretariat. Version: July 2018. `http://www.ietf.org/internet-drafts/draft-yeung-g-ikev2-14.txt`. 2018 (draft-yeung-g-ikev2-14). – Internet-Draft. – `http://www.ietf.org/internet-drafts/draft-yeung-g-ikev2-14.txt`