Secure Vehicle Communication

Deliverable 2.1-App.A

Baseline Security Specification

Project:	**Sevecom**
Project Number:	**IST-027795**
Deliverable:	**D2.1-App.A**
Title:	**Baseline Security Specification**
Version:	**v1.2**
Confidentiality:	**Public**
Author:	**Frank Kargl (editor)**
Date:	**10.04.2009**

Part of the Sixth Framework Programme
Funded by the EC -DG INFSO

Control Sheet

Version History			
Version number	Date	Main author	Summary of changes
0.1	16.04.2007	Frank Kargl	Initial version of the document
0.2	21.12.2007	Frank Kargl	Extended structure and included additional content
0.3	11.02.2008	All partners	integrated additional contributions
0.4	13.02.2008	All partners	peer review of components
1.0	14.02.2008	Frank Kargl	prepared version for submission
1.1	30.06.2008	Frank Kargl	draft for next version
1.2	10.07.2008	Frank Kargl	implementation basis
1.3	15.11.2008	All partners	revision based on implementation progress
1.4	31.03.2009	All partners	revision for review meeting
2.0	10.04.2009	Frank Kargl	final version for submission

Approval		
	Name	Date
Prepared	Frank Kargl	10.04.2009
Reviewed	All Project Partners	12.04.2009
Authorized	Antonio Kung	14.04.2009

Circulation	
Recipient	Date of submission
Project Partners	15.04.2009
European Commission	15.04.2009

Contents

1 General Overview

1.1 Introduction

The overall system architecture is described in SeVeCom Deliverable 2.1 and not repeated here. It is assumed that the reader is familiar of this document.

The SEVECOM baseline security architecture consists of multiple modules that each contribute a specific security functionality. The objective of this document is to describe all modules and components with a sufficient level of detail so that components and interfaces are consistently defined and the same level of details is provided for each mechanism. This involves naming conventions, consistent notations as well as a repository of names.

The specification of components follows the listed criteria:

- The specification should be complete enough so that the implementation does not need to take any design decisions (e.g. the fields and semantic of the certificate data structure should be described).
- The specification should separate the description from the mechanism (e.g. store the certificate) from comments on the mechanism (e.g. “the reason for doing this is to prevent man in the middle attacks . . . ”).
- The specification should not include unnecessary decisions that can be left to the implementer (e.g. format of internal data structures, or if we use an existing standard, explaining fields that are irrelevant to the mechanism).
- The specification should be maintainable, i.e. problems with the specification should be easily modifiable (e.g. do not need to read 20 lines of text to figure out where to make a change), and easily extendible (e.g. we should be able to change easily a section because there is a decision to change an algorithm).

Component descriptions can be created in an iterative process where the level of details provided is increased over time. Details for the creation process timeline are annotated in the component template.

1.2 Glossary

Here is an alphabetically sorted list of terms and acronyms used throughout this document:

API:	Application Programming Interface
CA:	Certificate Authority
CALM:	Continuous Air interface for Long and Medium distance
CRL:	Certificate Revocation List
DSRC:	Digital Short Range Communication
DMV:	Department of Motor Vehicles
ECDSA:	Elliptic Curve Digital Signature Algorithm
ECU:	Electronic Control Unit
GPS:	Global Positioning System
HSM:	Hardware Security Module
IVC:	Inter-Vehicular communication (equal to V2V + V2I)
ITS:	Intelligent Transport System
PKI:	Public Key Infrastructure
OBU:	Onboard Unit
QoS:	Quality of Service
RSI:	Roadside Infrastructure
RSU:	Roadside Unit
R2V:	Roadside to Vehicle
TOC:	Transportation Operation Centre
TCU:	Telematics Control Unit
TTL:	Time To Live
TESM:	Tamper Evident Security Module
VANET:	Vehicle Adhoc Network
V2V:	Vehicle to Vehicle communication
V2I:	Vehicle to Infrastructure communication
VC:	Vehicular Communication
VIN:	Vehicle Identification Number
VSCC:	Vehicle Safety Communication Consortium

1.3 Notation

Throughout this document, we use the following consistent notation rules.

General naming conventions	
X, Y, Z	network entities (no distinction between vehicles and RSU)
V	an unnamed vehicle
V_X, V_Y, V_Z	vehicles denoted as X, Y, and Z
RSU	an unnamed roadside unit
RSU_X, RSU_Y, RSU_Z	roadside units denoted as X, Y, and Z
$TComponent.method(Ua,$ $constVb)$	calling "method" in "Component" providing arguments a and b of types U and V. The return value is of type T
$ComponentName$, $longMethodName()$	names should adhere to the Java naming conventions
$\|$	concatenation
Cryptographic naming conventions	
K_{XY}	symmetric key shared between X and Y
PK_X	asymmetric public key of X, can also be used as signature verification key
SK_X	asymmetric private key of X, can also be used as signature generation key
$data$	generic data, e.g. message content
Cryptographic operations	
$byte[] = sign_{SK_X}(byte[]data)$	signature created by X over $data$
$bool = ver_{PK_X}(byte[]s, byte[]d)$	verifies if s is a valid signature over d created with SK_X

Table 1.1: Notation

2 Security Manager

2.1 Overview

The Security Manager is reponsible for overall system organization, instantiation and configuration of components, hooking of the security subsystem into the communication stack, and dispatching of (some) calls between components.

First of all, it is responsible for the configuration of the security components and for their instantiation. In addition, it acts as an interface between security components, mainly between the Secure Communication components (e.g. Secure Beaconing component) and the components managing credentials like ID-Management component, Pseudonym Application component etc.. One of the objectives is to encapsulate logically close operations that are used by multiple other components. This dispatching functionality both offers and requires various services to and from other security components.

2.2 Hooking Component

2.2.1 Purpose of component

The hooking component implements the Inter Layer Proxies (ILPs), which are used to realise the hooking mechanism as described in chapter 5.5 of the SeVeCom Deliverable 2.1. Therefore one or multiple ILPs are inserted between the layers of the existing network stack implementation. After registering a component to an ILP, incoming messages are passed to the component, which is able to modify or drop the message. If the message is not dropped, it continues travelling through the stack.

2.2.2 Interfaces and Services

The interface, a component has to implement:

```
virtual int eventHandler(C2xIlpMessageEvent* event, Message** message)
```

The method to register a component to an ILP:

```
bool registerHandler(C2xIlpEventHandler* client, C2xIlpMessageEvent* event)
```

2.2.3 Description

For registering a component to a ILP, it has to implement the eventHandler interface and to call the registerEvent method from the specific ILP instance. The registerEvent method takes two parameters, the first (C2XIlpEventHander*) is the object, which implements the eventHandler method that will be called in case of a specific event. The second parameter (messageEvent*) consists of a event object, that specifies the exact circumstances, on which an event shall be triggered. The event object specifies the layer, the queue (input, output, forward) and the message type that have to fit to a bypassing message, so that the eventHandler method is called. In this case the ILP in turn creates a event object that also contains the actual layer, the queue and the message type and passes it with the specific message to the eventHandler. Due to this approach, the eventHandler can distinguish between different situations, so it is possible to register one eventHandler at different ILPs or for different message types. The message which is passed is accessible via a double pointer, so the component can replace it and pass it back to the ILP. In addition, a return value is returned, which indicates whether the message was modified, if it should be dropped or passed to the following layer.

To execute a command or insert a self generated messages into the stack, the sendMessageUp or sendMessageDown methods of a ILP can be called, that directly pass the message to the next layer.

2.3 Configuration Component

2.3.1 Purpose of component

The security policies are defined in a configuration file which is read by the security manager at initialisation time. This file contains information for all the security components the security manager must configure.

2.3.2 Prerequisites

2.3.3 Interfaces and Services

Configuration file: For every security component, this file specifies:

- The name of the library that contains the binary code of the component,

- Optionally some parameters that must be passed to the constructor of the component,
- A list of types for incoming messages that concern this component. These types will allow the communication stack to know what incoming messages must be passed to the component,
- Types for outgoing messages that concern this component. This filters will allow the communication stack to know what outgoing messages must be passed to the component.

A BNF description of this file is given next.

```
file  ::= list-of-components
list-of-components ::= empty | component-desc | component-desc list-of-components
component-desc ::= component library filters links
component ::= '[' SYMBOL ']'
library ::= 'lib' '=' SYMBOL
filters ::= empty | filter | filter filters
filter ::= direction list-of-ids
list-of-ids ::= empty | id | id list-of-ids
links ::= empty | SYMBOL | SYMBOL links
```

2.3.4 Description

The tasks realised by the security manager at initialisation time are depicted in the figure below. First, it reads and analyses the configuration file. Second, for each component specified in this file, it loads its code into memory and instantiates it by calling its constructor eventually with its parameters (if provided). Third, it uses the types attached to the component for knowing what callback must be called for the messages which match these types. In order to do that, it calls the method getCallback of the component with the types it has just read as parameters. Forth, it registers the callbacks it has obtained at the previous step. For example in the figure below one can see that the method getCallback of the Beacon component has returned the callback B_send for the messages the types of which belong to the list types1. This callback is then registered to the stack at the right level. Further, each time a message the type of which is in the list types1 is to be sent, the stack calls the B_send callback. The same goes for the B_recep callback. For a component there can be as many number of list of types as necessary. Each list is associated with a callback.

2.3.5 Performance

2.3.6 Discussion

Security issues: Some precautions are necessary for this configuration process. The security manager must verify the integrity of the configuration file and of the binary codes of the components to load. Signatures can be used for these verifications. The latter will undertaken by the crypto module. Therefore this module must be already running i.e it must be launched before the security manager. Its code may be (or must be) stored in a TRSM (Tamper Resistant Security Module).

2.4 Dispatcher Component

2.4.1 Purpose of component

All security modules are registered. Each time a message is to be sent or is received, the corresponding callback (i.e. attached to this type of message) is called. This callback will then use the services of the security module in order to add signature for outgoing messages and verify signature for incoming messages.

2.4.2 Prerequisites

Dispatches methods among security components, and thus requires all components for which it is configured to relay calls.

2.4.3 Interfaces and Services

API for outgoing messages

```
public SMGR_return_code createMessageSignature(AbstractSecureMessage& msg);
```

In order to be to sure that the timestamp and the certificate are not altered before the insertion of the signature the 3 operations setTimeStamp, setCertificate and setSignature must be done in one single call. This is coherent with the deliverable 2.1 version 2.0 final (page 56). But this in contradiction with the baseline (page 26). For this reason the method calls sequentially these three methods in this order. Is the position of the vehicle also concerned by this problem ?

```
public checkPosition(AbstractSecureMessage& msg);
```

It checks the presence a valid position in the message. If no position is present it will ask the GPS module for a position and insert it in the message. The method can be implemented in the secure manager or in the network layer. In the second case the previous question is not accurate.

```
private TimeStamp getCurrentTime();
```

This method returns the current time out of the tampered evident security module.

```
private void setTimestamp(AbstractSecureMessage& msg);
```

This method calls the method GetCurrentTime and stores the result in the message. It returns OK in case of success and SMGR_NOCLOCK_DEVICE if the getCurrentTime method cannot return a time stamp.

```
private void setSignature(AbstractSecureMessage& msg);
```

This method calculates a signature for the message. For this operation a certificate is necessary. Therefore this method will call the getCertificate method. What certificate will be use for the signature ? long time certificate or pseudonym ? On what criteria this choice will be made ?

```
private Certificate& getCertificate(MessageBase msg);
```

This method returns a certificate to use for the message to send. It can be a pseudonym or a long term certificate. On what basis the choice will be made ?

```
private void setCertificate(AbstractSecureMessage& msg);
```

This method calls the getCertificate method and store the result in the message.

2.4.4 API for incoming messages

```
public SMGR_return_code verifyMessage(AbstractSecureMessage& msg);
```

On reception of a message a certain number of verifications are required for security purpose. These verifications are done by the following internal methods : verifyCertificate, verifyAuthentication, verifyTimeStamp (for Beaconing). It returns SMGR_OK in case of success and an error code otherwise. This error code is returned by the internal method which has detected the error. See below the descriptions of these methods.

```
private SMGR_return_code verifyCertificate(AbstractSecureMessage& msg);
```

This method verifies that the sender of the message is actually a valid participant of the network. It verifies that the certificate of the sender is valid and still in use (e.g not revocated). It returns SMGR_OK in case of success and otherwise it returns the following error code

- SMGR_INVALID_CERTIFICAT when the data does not correspond to a certificate or if it is ill formed.
- SMGR_REVOCATED_CERTIFICATE when the certificate corresponds to a revocated certificate.
- SMGR_EXPIRED_CERTIFICATE when the validity period of the certificate has expired.

```
private SMGR_return_code verifyAuthentication(AbstractSecureMessage& msg);
```

This method verifies that the identified sender has effectively sent the message. This verification allows at the same time to ensure that the message has not been altered. It returns SMGR_OK in case of success and the following error codes otherwise

- SMGR_AUTHENTICATION_FAILURE if the sender is not the expected one of if the message has been altered.
- SMGR_ILL_FORMED_SIGNATURE if the signature is ill formed

```
private SMGR_return_code verifyTimeStamp(AbstractSecureMessage& msg);
```

This verifies that the data is up-to-date i.e. it verifies that the timestamp is valid and is not outdated. It returns SMGR_OK in case of success and otherwise

- MSGR_INVALID_TIMESTAMP if the timestamp is found invalid
- MSGR_OUTDATED_MSG if the corresponding message if outdated.

Base type for all the message using secure communication

In order to secure messages, we need a common base class SecureMessage which provides the following methods : Access and modification of security related fields of the message.

- void setTimeStamp(typeTimeStamp& time) : for setting the timestamp field.
- void getTimeStamp(TimeStamp& time) : for getting the timestamp out of the message.
- void setCertificate(Certificate& cert) : includes the certificate cert in the corresponding field of the message.
- void getCertificate(Certificate& cert) : extracts the certificate out of the message.
- void setSignature(Signature& sign) : adds the signature in the message
- void getSignature(Signature& sign) : extracts the signature.

In order to make its job properly, the crypto support module needs a array. Therefore a function is needed for putting all the necessary information in a array before calling the crypto support module. This allows the cryptographic part to be independent of the physical arrangement of the data in the packet. Only the derivate class (SecureBeaconMessage for example) knows what data are covered by the signature, so the method need to be purely virtual.

- virtual void getSignedPart(UINT8 *p_signedData, UINT8 *p_length) = 0

2.4.5 Description

Outgoing messages

The steps that are taken to process outgoing messages:

1. a message is to be sent
2. the corresponding callback is called
3. the callback requests the security manager to add all information needed for security purpose
4. the security manager asks the pseudonym manager for getting a pseudonym
5. the security manager requests the crypto support module to add timestamp and signature
6. the result is returned

Before sending the message, the MAC address is changed in order to reflect the new pseudonym.

Incoming messages

The steps that are taken to process incoming messages:

1. a message is received
2. the corresponding callback is called
3. the callback requests the security manager to verify the signature
4. the security manager calls the trust management module in order to verify the certificate
5. the security manager calls the crypto support module in order to verify the message
6. the result is returned

3 Identification and Trust Management Module

3.1 Overview

The Identification and Trust Management Module defines components that are responsible for handling identities and credentials, with various operations including provision, renewal, or revocation of credentials.

3.2 Identification Management

3.2.1 Purpose of component

This component provides the means to uniquely identify communicating entities in vehicular networks, that is, vehicles or road side units at the wireless part of the network and trusted third parties at the wire-line part of the system. This component describes the details of public key creation and management.

3.2.2 Prerequisites

We assume there is an asymmetric cryptosystem, as defined in the cryptographic support module. For our purpose, the key pairs must be suitable for creation and verification of signatures.

Furthermore, we assume a suitable administrative process that will initialize identifiers for new vehicles or on-board units. During this process, names need to be assigned to entities and corresponding key pairs must be generated and installed in the vehicle.

The Crypto Support Module must provide for this initialization process the following function:

```
PublicKey init_device(Identifier ID)
```

Technical details regarding this operation are given in (Sec. 7). We do not define the related administrative procedures in detail, as manufacturing, installation, and operation procedures for OBUs are currently not foreseeable.

3.2.3 Interfaces and Services

The cryptographic keys described in this section will be used by other components, notably in components of the cryptographic support and secure communication modules. The private cryptographic key will only be accessible via the mechanisms described in Sec. 7.

This component defines a signature as follows:

```
typedef byte[] Signature;
```

In the current implementation this is a conversion of an ECDSA signature into a byte array. This array contains two numbers R and S. The length of the signature is 42 bytes for a 160 bits key (corresponding to an 80 bit security level).

This component does not directly specify any API.

3.2.4 Description

We denote the identifier of an entity X by ID_X. The long-term identity is represented by ID_X and is associated with a name of X, a cryptographic key pair (SK_X, PK_X), and a set of attributes of the entity.

The exact format of this unique long-term identity ID_X is not specified here, as it will be the outcome of an agreement between car manufacturers and authorities, similar to the use of Vehicle Identification Numbers (VINs). Such identifiers of the same format will be assigned both to vehicles (own or of other's) and road-side units.

Each identifier (and thus entity) is bound to an asymmetric key pair (SK_X, PK_X). A variety of asymmetric (public-key) cryptosystems is available; we recommend the use of Elliptic Curve cryptography (e.g. based on the IEEE P1363 standard). The exact form of implementation will also be influenced by an agreement of which parts of the standard (if not all) will be supported, and the characteristics of the Hardware Security Module (HSM).

The private key SK_X is generated by and stored in the HSM through the procedure detailed in Sec. 7.3. The public key PK_X is not stored in the HSM.

3.2.5 Performance

As this section does not describe direct operations to be used in VANETs, there are no significant performance implications of the mechanisms presented. However, depending on the cryptosystem and key length chosen, the use of the key pair for signatures or encryptions will largely influence the overall performance of all the other components.

3.2.6 Related Work

See standard IEEE P1363[1] and NIST "Recommended Elliptic Curves For Federal Government Use"[2] for details on Elliptic Curves Cryptography.

3.2.7 Discussion

The convergence to a format of a unique identifyer per system entity is beyond the scope of a security document, and it depends on a multitude of factors. The binding of such a unique identifier to unique cryptographic material enables entities to engage in secure transactions.

3.3 Trust Management Component

3.3.1 Purpose of component

This component describes the trusted third party (TTP) needed to provide the certificates that are necessary for the identification component described in Sec. 3.2. We denote here the TTP, referred to interchangeably as a *Certification Authority* or CA, by T.

3.3.2 Prerequisites

This component relies on the data structures defined in the Identification component (Sec. 3.2). It further assumes the provision of a revocation component (Sec. 3.4). Finally, for each certificate validity period E the availability of reliable communication channel between system entities and the TTP T is assumed for the time needed to perform the certificate update protocol.

[1] http://grouper.ieee.org/groups/1363/
[2] http://csrc.nist.gov/CryptoToolkit/dss/ecdsa/NISTReCur.pdf

3.3.3 Interfaces and Services

This component issues certificates as defined in Sec. 3.2:

```
struct {
        Identifier ID;
        public_key publicKey;
        Attribute[] attributes;
        Timestamp StartOfValidity;
        Timestamp EndOfValidity;
        Identifier Issuer;
        Signature certificateSignature;
} Certificate;
```

The length in bytes of the certificate is the sum of lengths of all element in the structure. Bellow are the lengths for each field in the current implementation. We consider sizeof(unsigned int) = 2, sizeof(unsigned char) = 1 and sizeof(long) = 4.

- length(Identifier) is 2, because Identifier an unsigned int;
- length(Attributes) is 2, there are 2 elements of one byte as mentioned above
- length(Timestamp) is 8, there are 2 longs, one for seconds and one for useconds
- The Public Key is composed of three big numbers. The representation of these numbers in bytes is the same as for the signature, the length of the number is writen first then the bytes of the number follow. So the total length is $3 + len_{no1} + len_{no2} + len_{no3}$. While we are working only with 2 deminsional points $len_{no3} = 0$, we ignore this number and the lenght is gets equal to $2 + len_{no1} + len_{no2}$.
- length(Signature) depends on the security level used and folows the formula $2 + len_R + len_S$.

Summing up all this values and considering a key of 160 bits (represented into 42 bytes) the Certificate length is of 122 bytes.

This component implements a certificate store, which supports retrieval of known certificates either giving the ID of the entity or the key identifier as an argument. Other methods allow the storage of new certificates, checking of validity of certificates, etc:

```
/* retrieve cert from store using entity ID */
Certificate getCertificate(
    Identifier ID
    )

/* retrieve cert from store using key ID */
Certificate getCertificate(
```

```
    String keyIdentifier
    )

/* provide new cert, save in cert store if new, verify validity */
Status verifyCertificate(
    Certificate cert
    )

/* verify that valid certificate for entity is known */
Status verifyCertificate(
    Identifier ID
    )

/* verify that valid certificate for key is known */
Status verifyCertificate(
    String keyIdentifier
    )

/* save certificate in cert store */
storeCertificate(
    Certificate cert
    )
```

This certificate store allows accessing own certificates as well as certificates of other parties.

The renewal of the own long-term certificate is achieved via a two-party protocol executed by system entity X and in particular its hardware security module (HSM) and T.

The update protocol is defined as follows:

Certificate Update Protocol			
Purpose:		Retrieving a new certificate with extended validity before the previous one expires.	
Parties:		Vehicle V_X and CA T.	
Pre-condition:		Existing certificate is close to expiration but still valid	
Post-condition:		After a successful run, the entity has received a new certificate for the new public key. The new key pair and certificate should then be used at the beginning of their validity period. In case of failure, the devices can retry as long as the old certificate is still valid. After that, an automatic renewal is not possible any more, and a new certificate must be installed manually.	
	Direc.	**Message**	**Description**
1	$V \rightarrow T$	$T, ID_X, PK'_X, info, sig_{PK_X}$	V sends signed request for new certificate for key PK'_X to T
2	$V \leftarrow T$	$cert_{PK'_X}$	T returns new certificate
3	$V \rightarrow T$	$T, ID_X, info, sig_{PK'_X}$	T acknowledges receipt of updated certificate

3.3.4 Description

The TTP can be operated by local authorities, for example, at a region, state, or country level; or it can be operated by an international organization. T has a policy that determines that any newly manufactured car equipped with an onboard unit or roadside unit, or any onboard that is manufactured for retrofitting, will receive one certificate.

The trusted third party, T, creates a certificate of the form:

$$cert_{PK_X} = (ID_X, PK_X, A[], E, T, sig)$$

where ID_X and PK_X are the identity and public key of the entity X, T is the identity of the TTP, $A[]$ is an attribute list by which T declares its trust in certain attributes of ID_X (and thus X), and E is the validity period of the certificate, determined by two time-stamps, a start and an end time. The generation and binding of all this material by the TTP is achieved by the sig field, a signature calculated by the TTP T over the rest of the certificate data, as follows:

$$sig = sign_{SK_T}(ID_X, PK_X, A[], E, T)$$

Each device ID_X is initialized with a certificate using a trustworthy communication link to T (which is assumed to be available during manufacturing). To obtain credentials (certificate) and cryptographic material at a later point in time, a two-party protocol with T is used.

At a point in time τ time units before the expiration of its certificate, X's HSM generates a new key pair (SK'_X, PK'_X). It then generates a Certificate Request (CR) for T, providing the identity of the TTP, the entity's identity ID_X, its new public key PK'_X), and T-specific information in a string $info$; we denote this as $CR_{data} = (T, ID_X, PK'_X, info)$.

Then, ID_X signs the CR_{data}, where the request is uniquely identified by the $info$ string, which includes a nonce, i.e., a not-previously-used identifier with respect to T and ID_X, and the new validity period E'. The request sent

$$CR = (CR_{data}, sign_{SK_X}(CR_{data}))$$

If the request is authenticated and the ID_X has not obtained yet a certificate for the period E', T generates a new certificate:

$$cert_{PK'_X} = (ID_X, PK'_X, A[], E', T, sig)$$

with its fields defined above. T sends this certificate to ID_X as a Certificate Response (CRS). The protocol concludes with the transmission by ID_X of a Certificate Acknowledgement (CAck):

$$CAck = (T, ID_X, info, sign_{(SK')_X}(T, ID_X, info))$$

Upon reception of the CAck, T considers the installation of the new certificate successful. Note that the duration of the E' is a system parameter, depending on the T, and unless the value provided in CR is compliant with the system operation, a new certificate will not be provided. [3]

The CA provides a remotely accessible interface for certificate renewal, performed as defined above. Using this interface, vehicles that reach the end of their certificate lifetime can generate a new asymmetric key pair, send the new public key to the CA, authenticated by the currently available and valid cryptographic key, and receive a new certificate.

The presence of the HSM ensures that the vehicle will not utilize the newly acquired certificate and the corresponding private key during the validity period E of its current key and certificate. This is so, as E and E' are partially and to a small extent overlapping, exactly to enable the renewal.

[3] Note that in the case of retrofitting, that is, installation of an OBU in used cars, an off-line identification process is necessary to ascertain the correctness of used attributes (e.g., physical or other attributes).

The CA certificate is always valid, in the sense that the CA itself ensures the distribution of a new certificate when necessary. The CA certificate format follows that of the certificate formats for the entities:

$$cert_{PK_T} = (T, PK_T, A_{CA}[], E_{CA}, I, sig)$$

where T is the unique identity of the CA or TTP we denote as T[4], PK_T is its public key, E_{CA} is the validity period of this certificate, $A_{CA}[]$ is a list of attributes for T (such as the geographic area it covers), and I the identifier of the certificate issuer. E_{CA} is significantly larger than any E for the validity of entity certificates. The issuer of such certificates for a CA is either a hierarchically superior authority R, or in the case of several independent authorities operating in the absence of an R, $I = T$. In that latter case, to ensure inter-operability and enable secure communication between vehicles registered with distinct authorities (trusted third parties), each trusted third party S can generate a certificate for each trusted third party $T \neq S$ if CA policies are compatible. This is essentially the concept of *cross-certification*. In that case, for example, $I = S$ above, and

$$sig = sign_{SK_S}(T, PK_T, A_{CA}[], E_{CA}, S)$$

3.3.5 Performance

The performance overhead due to this component is not considered in detail and it is very low for the system entities: the certificate renewal is infrequent, with certificate validity periods being a significant fraction of the vehicle and on-board unit. These long-term credentials are not used for inter-vehicle or vehicle-to-road-side-unit communication. As such, they are used for infrequent transactions, as those in Sec. 4.2.

3.3.6 Related Work

The organization of a certification authority for the Internet has been considered, and a standardized architecture currently in broad use is defined in the PKIX documents of the Internet Engineering Task Force (IETF) (RFCs 3279, 3280, and others).

[4] for simplicity, we avoid here the distinction between T and ID_T

3.3.7 Discussion

The exact form of a certification authority is beyond the scope of this document and project. This is so, because organizational and policy issues, among other factors that do not relate only to the technically appropriate solution, will affect the choice of a specific form. It is also possible that a mix of forms will be adopted as the deployment of vehicular communication systems takes place. For example, it is not clear which entity will be the one that certifies all trusted third parties, or if one will exist, and what portion of the system it will cover.

3.4 Revocation Management

3.4.1 Purpose of component

This component provides the means to revoke long-term identification and related cryptographic keys and credentials of entities in vehicular networks before expiration of the certificate lifetime, and this way essentially evict the entity from the network. This component only deals with revocation of long term credentials and not pseudonyms as discussed in Sec. 4.

With the revocation mechanism described here, vehicles do not receive revocation information directly. Instead, revocation is done indirectly by making a pseudonym provider rejecting new pseudonyms to vehicles with revoked long-term credentials.

3.4.2 Prerequisites

The presence of trusted third party or certification authority is assumed, along with the means to identify entities, as detailed in Sec. 3.2 and Sec. 3.3.

3.4.3 Interfaces and Services

The process relies on an interface between the Pseudonym Management Component at the side of the Pseudonym Provider (PP), in Sec. 4.2, and the trusted third party. As this interface is not part of the vehicular communication system, it will currently not be described here.

3.4.4 Description

A node deemed illegitimate (e.g., expired registration or stolen) or malfunctioning can be removed from the network. This is possible by revoking the long-term credential of the vehicle. Revoking the long-term credentials of a node evicts the node from the system, but it does not automatically prevent the node from participating in the VC system operation. This is so, because the long-term identity and credentials are not utilized for communication.

However, long-term credentials are used for the vehicles to obtain the credentials they utilize for communication, as specified in Sec. 4.2: the pseudonym provider will supply short-term credentials only to legitimate members of the system, i.e., registered with a CA. Once the pseudonym provider is notified that a specific node is revoked, it will not respond to any further pseudonym requests, as specified in Sec. 4.2. This will prevent the illegitimate node from any further participation in the system after expiration of its short-term pseudonyms.

Trusted third parties compile revocation information in the form of Certificate Revocation Lists (CRLs), which are of the form:

$$(ID_X, ID_Y, ID_W, \ldots), T, time, SEQ, sign(SK_T, (ID_X, ID_Y, ID_W, \ldots), T, time)$$

A list of identifiers of vehicles[5] to be revoked is followed by the CRL issuing authority identity T, a validity period of the CRL $time$, a CRL sequence number SEQ, and the given issuer T. The distribution of revocation lists among distinct CAs (TTPs) facilitates the eviction of entities throughout the system, if, for example, a vehicle tried to engage in communication in an area administered by a CA other than the one that certified its original (and valid till the revocation) credentials.

The procedure described above is only applicable to vehicles using short-lived pseudonyms. As RSUs are not expected to use pseudonyms, but instead communicate using a their long-term credentials. A dedicated revocation of RSU certificates is not implemented, instead the life-time of RSU certificates should be kept very small (e.g. one day). As RSUs are assumed to have a wire-line or other infrastructure-based network connection to T, they can run the update protocol described in Sec. 3.3[6] more often than vehicles. If overhead of this frequent renewal is a concern, [KSW06] describes a mechanisms to reduce that overhead.

[5] depending on the capability of CRL users to map identifiers to public keys, the list could additionally contain the corresponding public keys

[6] Note that RSUs are also equipped with an HSM.

3.4.5 Performance

As the revocation of long-term identifies and the distribution of CRLs is done exclusively between CAs (TTPs) and pseudonym providers with which the vehicle may then perform transactions, there is no revocation specific communication in the vehicular part of the network. Therefore the size of CRLs and the frequency of distribution are not affecting the performance of the vehicular network.

3.4.6 Related Work

The concepts of revocation are broadly in use in a variety of settings, in the Internet and beyond (e.g., credit cards usage). In our case, the communication between the CAs (TTPs) and the service providers can be performed by the standard TCP/IP protocol stack using PKIX protocols.

3.4.7 Discussion

The size of CRLs is an aspect that the designers of this architecture cannot know at the time of development. At worst, the scale of CRLs will be linear in the number of network entities. However, the number of revoked vehicles would be reasonably expected not to exceed a small fraction of the registered vehicles. As such, the size of the CRL will follow at most the number of vehicles registered during the average certificate lifetime. As CRLs are distributed across the wireline and not the vehicular part of the network, no additional performance consideration is made here, assuming ample bandwidth and processing power.

We currently assume no need for revocation of pseudonyms, as their expiration time is considered to be very short. Therefore it is more efficient to simply wait for their expiration instead of creating additional revocation overhead. If this will be nevertheless necessary, [PMH08, RPJP06, KSW06] provide approaches how to organize efficient revocation of pseudonyms in vehicular and ad-hoc networks.

4 Privacy Management Module

4.1 Overview

The Privacy Management Module provides privacy-enabled vehicular communications while still fulfilling necessary secure requirements. It leverages on resolvable pseudonyms (i.e., certified short-term public keys) to achieve a defined level of privacy for individual vehicles and yet allow the identification of valid vehicles and public bodies to resolve pseudonyms to vehicle long-term identities under well-defined conditions. Vehicles are loaded with sets of pseudonyms they can then use for message signing. By switching pseudonyms and using them only for a limited amount of time, privacy attacks are impeded significantly, notably in the presence of an adversary (eavesdropper) that does not cover the entire network area.

4.2 Pseudonym Management Component

4.2.1 Purpose of component

The Pseudonym Management is responsible for administration of the on-board pseudonym pool. The functionalities of this component include the initiation of pseudonym generation, adding and deleting pseudonyms in the pool, monitoring the pseudonym usage status, and the configuration the pseudonym refill policy.

Besides, this component also defines the structure of pseudonyms, which are *short-term certified public keys* that do not provide additional identifying information. They can, however, include a list of vehicle attributes that need to be attested by a TTP. These attributes should be sufficiently generic so that an identification of an individual vehicle is not possible based on the attribute list.

4.2.2 Prerequisites

This component requires each vehicle to have obtained a unique long-term identifier ID_X which bounds to a corresponding certificate $cert_{PK_X}$ provided by the Identification & Trust Management Module 3.

It is prerequisite that there is a PP in the Public Key Infrastructure (PKI), who can certified public keys from individual vehicles (i.e., issue pseudonyms to legitimate vehicles), and such pseudonyms are trusted by all nodes in the network.

Additionally, it is assumed that there exists a Hardware Security Module (HSM) where secret keys from pseudonyms can be stored securely and that will do all secret key operations.

4.2.3 Interfaces and Services

The Pseudonym Management component communicates with a remote Pseudonym Provider using a bi-directional protocol to acquire new pseudonym certificates for locally generated pseudonym keys.

We define the data structure for a pseudonym as:

```
struct {
          Identifier ID;
          public_key publicKey;
          Attribute[] attributes;
          Timestamp StartOfValidity;
          Timestamp EndOfValidity;
          Identifier PseudonymProvider;
          Signaure certificateSignature;
} PSNYM
```

It can be noticed that the PSNYM structure is equivalent with the Certificate structure. As a consecqunce if the same security level is used the PSNYM's length equals the Certificate's length. If we take again a security level of 80 bytes (public key of 160 bits) the total legth of this PSNYM is 122 bytes.

ID can be used to uniquely identify a private/public key pair (in other words, the pseudonym and its corresponding private key) within the modules and components in the security architecture. The structure of pseudonym is similar to the certificate defined in Identification and Trust Management Module. However, any information that can be used to identify a specific vehicle is removed from the certificate to form the structure of the pseudonym.

The following also defines the services and interfaces in this component.

```
int[]  getPseudonymPoolStatus()
```

This method returns the status of the pseudonym pool. There are two elements in the array $int[]$, i.e., the number of *used* pseudonyms and the number of *un-used* pseudonyms.

```
PSNYM[] generatePseudonyms (
                        int NumberOfPsedonyms;
                        )
```

This method initiates a request to the Identification and Trust Management Module for generation of a new set of pseudonyms. Notice that the pseudonym generation is a cooperative process between the HSM and the Identification and Trust Management Module. The actual generation is not done in this method (see Pseudonym Generation in Sec.4.2.4). The function returns new pseudonyms specified in the NumberOfPseudnyms.

```
boolean setRefillPolicy (
                 int maxNumberOfPseudonym;
                 int minNumberOfPseudonym;
                 int maxValidityPeriod;
                 int minValidityPeriod;
                )
```

This method sets the parameters for a vehicle's pseudonym refill policy. $maxNumberOfPseudonym$ specifies the maximum number of pseudonyms a vehicle can refill at one time. Therefore, $maxNumberOfPseudonym$ depends on the size of the on-board pseudonym pool, as well as the maximum allowance set by the pseudonym provider. When the number of *un-used* pseudonyms in the pseudonym pool is below the number specified by $minNumberOfPseudonym$, the on-board system should either prompt the driver to contact the pseudonym provider for pseudonym refill, or start the pseudonym refill if an automatic pseudonym generation mechanism is available (depending on the connectivity to infrastructure network). The next two arguments, $maxValidityPeriod$ and $minValidityPeriod$ specify the maximum and minimum validity period of each pseudonym in terms of hours or days (depending on actually implementations). Although a vehicle can set the value of the validity period, it should be in the range specified by the pseudonym provider. By specifying the range of minimum and maximum validity periods, the pseudonym provider can mandate the period of time for which the vehicle should refill its pseudonyms, while giving a vehicle a certain degree of flexibility to decide the frequency to request for pseudonym refill.

4.2.4 Description

Pseudonym

Pseudonyms are *a set of distinct certified public keys that do not provide additional identifying information*.

Instead of using a long-term identifier for signing messages, each vehicle is equipped with a set of short-term identifiers (e.g., $\{psnym_{X1}, \ldots, psnym_{xk}\}$) that consist of a key pair and corresponding certificates. Those pseudonyms are similar to the long term identifiers from Sec. 3.2 with the exception that they do not include any information which identifies an individual.

The pseudonym key pair of vehicle X $(PSNYM - SK_{Xi}, PSNYM - PK_{Xi})$ is a key pair as defined in Section 3.2. A pseudonym certificate for this key pair has the following format:

$$psnym - cert_{Xi} = (PSNYM - PK_{Xi}, A[], E, PP, sig)$$

where $A[]$ is the attribute list by which PP declares its trust in certain attributes of X, E is the validity period of the certificate, determined by two time stamps, a start and an end time. PP denotes the pseudonym provider which created this pseudonym certificate and sig is a signature over the certificate data calculated using PPs private key:

$$sig = sign(SK_{PP}, PSNYM - PK_{Xi}, A[], E, PP)$$

The pseudonym provider PP is an authority issuing pseudonym certificates similar to the CA described in Sec. 3.3.

When using pseudonyms, a safety-related message from a vehicle X will roughly have the following format:

$$MSG = data \mid sig \mid PSNYM$$

Details of message formats are given in the respective components described in the components of the Secure Communication Module.

Pseudonym generation

The generation of new pseudonyms basically involves two steps:

1. **Generation of key pairs:** the HSM module generates a set of new key pairs $(PSNYM - SK_{Xi}, PSNYM - PK_{Xi})\ \forall i = 1 \ldots n$.
2. **Retrieval of certificates:** the OBU contacts the pseudonym provider, sends all the pseudonym public keys plus authentication information, and in turn receives one certificate per key pair.

The communication with the PP has to be done via an authenticated and confidential communication link. The exact details of the pseudonym provider protocol where a PP communicates with vehicle X are as follows:

$$\begin{array}{lcl} X \rightarrow PP & : & PSNYM - PK_{X1}, PSNYM - PK_{X2}, \ldots, PSNYM - PK_{Xn} \\ PP \rightarrow X & : & N \\ X \rightarrow PP & : & sign(PSNYM - SK_{X1}, N), sign(PSNYM - SK_{X2}, N), \ldots \\ PP \rightarrow X & : & psnym - cert_{X1}, psnym - cert_{X2}, \ldots \end{array}$$

Pseudonym certificates are only issued, if vehicle X was previously able to authenticate to PP using ID_X and if X can prove knowledge of the corresponding secret keys $PSNYM - SK_{X1}, N$ by signing a nonce N.

Pseudonym Storage

The pseudonym certificates are stored in the OBU, the secret keys are stored in the hardware security module. Certificates are transmitted together with packets sent and contain only public information. Therefore, they do not need to be protected. Knowledge of secret keys of other vehicles' pseudonyms, however, would allow impersonation of those other vehicles and need to be stored in a protected way.

The storage space allocated for pseudonyms in the OBU depends on the available space in the OBU and the frequency of pseudonym usage (i.e., how often a vehicle needs to sign the messages with the same private key and the corresponding pseudonym).

Pseudonym Refill

Frequent change of pseudonyms used in communication ensures the privacy of vehicles. The Pseudonym Management should constantly monitor the number of valid pseudonyms in the storage, how often/long they have been used, and when they will expire. When the number of available pseudonyms falls below a certain threshold, the Pseudonym Management will restart the pseudonym generation procedure and obtain a new set of pseudonyms. This is called 'Pseudonym refill'.

Pseudonym Resolution

When issuing pseudonym certificates, the pseudonym provider PP stores the mapping between ID_X and all $PSNYM - PK_{Xi}$ at a pseudonym resolution authority PRA. The mapping is stored for the lifetime of the pseudonym plus some extra time. Based on certain legal conditions, public bodies, for example, vested with the power of law enforcement, can contact the PRA and request a resolution for any $PSNYM - PK_{Xi}$.

The PRA will then provide ID_X which essentially reveals the exact identity of the vehicle.

Received Pseudonyms Storage

A pseudonym received for the first time with a secured message, is verified and then stored with the result of the verification. When it is received again, for time optimization reason, the pseudonym is not verified again but the stored result is used. This storage is done in OBU and the allocated storage space depends on the available space in the OBU. The storage is refreshed, so expired pseudonyms are deleted.

4.2.5 Performance

Cryptographic operations with pseudonyms are not different from operations in the Identification & Trust Management module. The regular refill of pseudonyms creates both a communication and storage overhead that the vehicular communication system must be able to bear. However, as the change intervals and number of pseudonyms can be adjusted to the bearable overhead, there is bias between privacy and overhead that can be controlled by system designers/administrators.

4.2.6 Related Work

See “Security without identification: Transactions to make big brother obsolete” [DC85] for the concept of pseudonyms. Concept of pseudonyms in vehicular networks can be found in “Privacy Issues in Vehicular Ad Hoc Networks” [FD05]. A more detailed description of implementation of pseudonyms in vehicular networks is in “Architecture for Secure and Private Vehicular Communications” [PBH+07].

4.2.7 Discussion

In this component, we assume that pseudonyms are generated by the pseudonym provider, which is a particular instance of CAs within the PKI. A vehicle is thus required to connect to the infrastructure on a regular basis in order to obtain pseudonyms. As discussed, this creates a certain overhead. An alternative approach is to let vehicles to generate pseudonyms themselves on-the-fly as described in [CPHL07]. However, as this is an area of ongoing research and as the baseline architecture is meant to provide solid and well-understood mechanisms only, we use the concept of a pseudonym refill here and will provide more complex mechanisms in future components.

4.3 Pseudonym Application Component

4.3.1 Purpose of component

The Pseudonym Application provides pseudonyms which are used in secure communications. Beside providing valid pseudonyms, this component decides how long a pseudonym is allowed to be used in the communications and when to change to another pseudonym, according to the privacy policy defined either by the user or the VC system. It specifies a framework, on which a vehicle's decision of pseudonym changes are based. The component is also responsible for coordination of changes of identifiers in other layers in the communication stack.

4.3.2 Prerequisites

This component assumes that a vehicle X has a set of valid pseudonyms available. Details about format and creation of pseudonyms is given in the Pseudonym Management component (see Section 4.2). It is also assumed that the lower layer protocols, e.g., at the data link layer, can respond to the changing address command and update their addresses accordingly if possible.

4.3.3 Interfaces and Services

The Pseudonym Application component provides valid pseudonyms to other modules and components in the architecture (e.g. the components of the Secure Communication Module), and monitor the usage of pseudonyms in terms of duration. This component also coordinates changes of identifiers in the Communication Stack (e.g., IP address in the network layer, MAC address in the link layer) by sending a changing command to the Communication stack when pseudonyms change.

This component has the following services and interfaces:

```
boolean setPrivacyPolicy(
                double FixInterval;
                double RandomInterval;
          )
```

This method configures the overall pseudonym change policy (e.g., change every minute, every hour etc.) It configures two important parameters for the pseudonym change interval: the fixed interval τ and the random variable δ. The method returns true if setup success and false when setup fails.

```
PSYNM getCurrentPseudonym ()
```

This method returns the currently used Pseudonym.

```
boolean updateStackAddress ()
```

This method sends a changing address command to the communication stack. The TCP/IP stack and the MAC layer in the communication stack should change their addresses (i.e., IP address, MAC address) simultaneously after received the command. This method returns true if it succeeds, false if the communication stack is not able to change the addresses.

4.3.4 Description

The Pseudonym Application component provides a method which returns the *key id* of the pseudonym that is currently to be used. This key id is then to be given as an argument in calls to the HSM requesting e.g. signing of data. Based on this information, the HSM can then select the proper key material. The signature of this method is:

$$PSYNM, ID_{key} = PseudonymApplication.current();$$

For privacy reasons, each pseudonym will only be used in the communication for a short period of time and then discarded. The time-lapse since a pseudonym is used is denoted as τ. We suggest that τ is in the range of seconds to a few minutes.

After providing the pseudonym, Pseudonym Application sends a $change\_address((< layer >, < address >), \ldots)$ command to the communication stack instructing it to change the addresses on the respective layers. The communication stack will reset the current used identifiers as soon as all messages currently queued in the stack[1] have been sent.

The decision to switch to a new pseudonym is based on how long the current pseudonym has already been used (t_{used}). This time is calculated as the current time $t_{current}$ minus the time of the last pseudonym change t_{last}, i.e. $t_{used} = t_{current} - t_{last}$. The Pseudonym Application component compares t_{used} with the pseudonym changing interval that consists of a fixed time interval τ randomized by a random variable δ, and switches to a new pseudonym, when the following condition is satisfied:

$$t_{used} \geq \tau + \delta$$

[1] i.e., having already left the Secure Communication Module but having yet to be send by the communication stack

where δ is in a configurable interval $[-\epsilon, +\epsilon]$. This randomization is necessary as otherwise pseudonym changes could easily be tracked based on the pseudonym change timing observed by an attacker.

4.3.5 Performance

As a pseudonym change is a pure internal activity, there is no direct influence on performance. Note however, that our research has shown that change of short-term identity may influence the performance of certain network protocols [SKS+06].

4.3.6 Discussion

The current version specifies a very straight-forward algorithm for the decision of pseudonym change. More sophisticated algorithms could be envisioned that e.g. take the vehicle context (e.g., location, velocity, and neighboring conditions etc.) into consideration. Such approaches will inevitably increase the computation and implementation cost, and their effectiveness has yet to be evaluated. Future versions of Pseudonym Application components might address these issues.

Network operation considerations such as communication with an access point through the TCP/IP stack might require that the vehicle keeps the identifier fixed for a certain time. Under such situations, the pseudonym as well as identifiers in Communication Stack should remain unchanged. This can be achieved by a call from the communication stack to the Pseudonym Application component that prevents pseudonym changes for that time.

5 Secure Communication Module

5.1 Overview

The Secure Communication module deals with aspects of assuring security of the communication network. This means that the module takes care of the actual communication processes in order to ensure their reliability. For that, it also utilizes and cooperates with practically all other modules.

As components, we distinguish between several, typical types of communication in vehicular networks, which all have particular properties in terms of security. The component descriptions are somehow more generic than the other modules, as parts of these components need to be specifically tailored to the communication stack used. Details on this and the adaption layer can be found in the SeVeCom baseline architecture (SeVeCom Deliverable 2.1).

5.2 Secure Beaconing Component

5.2.1 Purpose of component

Assumptions on Communication Model

In vehicular networks, beaconing denotes a mechanism which broadcasts information periodically in the wireless transmission range of a node. Besides some identifier, the information typically includes the vehicle's own position and additional information like speed or heading. Beacons are usually not forwarded, i.e. are consumed after one hop.

Applications and also other stack components like geographic routing use beacons e.g. to determine the locations of vehicles in the vicinity. Using the location updates of vehicles around, an application inside one vehicle is able to predict the others' trajectories and thus is able to help with lane merging, for example.

Security Goals

As a basic goal, a receiver needs to be able to verify authenticity and integrity of beacons. This means that a vehicle must be able to trust in the content of a beacon message in a way that

- the sender is actually a valid participant of the network (e.g., a vehicle, RSU, traffic sign, etc.)
- the identified sender has sent the message, not another one
- the data is up-to-date
- the data has not been altered

5.2.2 Prerequisites

The component requires access to several services from other components. In particular, it relies on an established identity and credentials management, which allow for getting current credentials or even to have basic operations done there. For instance, the component does not care about the current identifier of a vehicle, its credentials or implemented cryptographic functions – it just uses the data and functions supplied in the corresponding components (see Section 3).

If pseudonyms are in use as described in the Privacy Module (see Section 4), the way of signing and verifying messages does not change - the privacy and identity management modules have to take care which keys to use.

In summary, the requirements for secure beaconing include:

- A mechanism to determine the current identity
- A mechanism to sign data
- A mechanism to verify a signature
- A mechanism to get the current time

5.2.3 Interfaces and Services

Hooking

The component provides an interface which allows the component to be hooked into the process of beaconing. Therefore, the component is notified when beacon packets are to be sent or received.

The secure beaconing component requires total control over the further processing of packets. For example, the secure beaconing component may figure out that an incoming beacon message is invalid and therefore decides to discard it. This can be done either by this component itself, or by the network layer implementation. The interface design here assumes that the network layer is able to process certain return codes and has to drop a packet accordingly, if the secure beaconing components indicates so.

5.2.4 Description

During the setup of the system, the secure beaconing component is hooked into the data delivery path. When we assume network layer beaconing, the Secure Beaconing component is attached between the network and link layer. Using this hook, the secure beaconing component will process the beacon data both upon sending and upon reception of a beacon.

Outgoing beacons:

When a beacon message is lined up to be sent, the hook redirects the message to the secure beaconing component.

To be able to scan the content of the beacon, the message format must be known to the secure beaconing component, at least to some extent.

As mentioned earlier, typical beacons will include at least

- the current vehicle identifier (pseudonym) X
- the current vehicle location loc_X

In addtion, the secure beaconing also requires a current time stamp t_c to be included in the beacon message in order to be able to ensure freshness of beacons.

For both efficiency and security reasons, these fields should not be duplicated in a beacon message. Hence, the implementation has to reuse existing fields. In case that the required fields are not included already, they have to be appended by the secure beaconing. Moreover, even if the required fields are already included, secure beaconing has to ensure that they comply with the security requirements. For instance, if the application has already added the field for the vehicle position, but this position information is not accurate enough for security reasons, another, appropriate location has to be appended by the secure beaconing.

Finally, the $PAYLOAD$ should contain:

$$PAYLOAD = X \mid loc_X \mid \ldots$$

After these preprocessing steps, the component uses signing capabilities of the identification and trust management module. Moreover, the current time t_c is returned together with the signature, as the Hardware security module provides a function to sign with timestamp.

After that, the beacon message will comprise payload, timestamp, signature and certificate:

$$BEACON = PAYLOAD \mid t_c \mid sig_{SK_X}(PAYLOAD \mid t_c) \mid cert_{PK_X}$$

The signed $BEACON$ will then be returned into the data delivery path.

Incoming beacons:

When a beacon arrives at a vehicle, it is passed over to the secure beaconing component via the hooking interface. The component will first check the attached signature by using the $verify$ method of the identification management module. If the signature can be verified, further post-processing is applied, like the freshness check to prevent replay of old messages. If the signature is invalid, the message is either discarded immediately or marked as invalid by the component. The choice depends on whether applications also want to process invalid packets and should be configurable.

After the signature check, which includes a certificate check, it can be guaranteed that

- The message was sent once by the given sender X
- The message has not been altered
- The sender is a valid network participant

Moreover, as the messages must not be replayed from vehicles passing by earlier, the freshness check needs to validate that the message's time stamp is recent. This requires a means to determine the current time, which is provided by the hardware security module.

Note that the freshness check should explicitly tolerate propagation delay. An allowed deviation of several seconds seems reasonable to prevent large-scale replay. This treatment also has the advantage that clocks do not need to be tightly synchronized. Nevertheless, if an older message is received, it is discarded.

Secure Beaconing API

Data structures

We define the following data structures that we use in the specification of the API. Note that further details on most parts of the structures are available in other part of the specification or are subject to the implementation of the network stack. For example, the format of a node identifier is left opaque here.

```
Packet {
  Identifier source
  Location   senderLocation
  <Optional Headers>
  <SecurityHeader>
  Data       data
}

SecurityHeader {
  Signature   sig_SKX
  Certificate cert_PKX
  Time        t_c
}
```

The following functions are hooked into the communication flow:

send

```
    inputs:
        Packet p
    outputs:
        SecurityHeader secHeader;
        uint ReturnCode;
    exceptions:
        InvalidPacket
```

Sends a beacon message. Time, signature, and a certificate is added to the Security-Header and returned to the communication stack to be appended to the message.

receive

```
inputs:
    Packet p
outputs:
    uint ReturnCode;
exceptions:
    InvalidPacket
```

Checks if a recently received beaconed message is correct. Returns 0 if the packet was correct otherwise returns an appropriate error code like certificate error, signature error, timestamp error. Note that the Packet here contains the SecurityHeader.

5.2.5 Performance

Due to the periodicity of beacon messages, performance is a relevant factor. Particularly, expensive cryptographic operations might influence performance. For example, if a vehicle has 50 neighbors in its wireless transmission range, and beacons are sent with 10 Hz, every vehicle needs to verify 500 signatures per second. Note that this node density and this beacon frequency are not unrealistic assumptions. 10 Hz is one of the proposed number for beaconing frequency and 50 vehicles within the transmission range are easily reached in a traffic jam. Assuming 3 lanes, a vehicle every 25 m and a transmission range of 300 m, which are not too pessimistic values, we get theoretically 72 vehicles within the transmission range.

As vehicle on-board units often do not possess extensive computing power, this needs to be respected. Because asymmetric crypto operations are relatively costly compared to symmetric operations, signing every beacon with such mechanisms may become inadequate. However, as computing power is variable and very likely to be improved in the future, the specification assumes to be able to process all signature verifications in time.

Future specifications of this component will include performance optimized versions of this component that will e.g. not need to verify signatures in every packet received.

5.2.6 Related Work

Several researchers already addressed the security of beaconing in vehicular networks. For instance, Hu and Laberteaux proposed a TESLA-based approach to reduce the number of required verifications [YCHKPL06]. However, this comes to the cost of delaying messages for one period of time and it requires a tight time synchronization (which could be affordable because of available GPS time).

A more general proposal to beaconing can be found in [RCCKL06]. Here, the authors deal with the problem that potentially many applications will use beaconing-like communication. Both periodicity and payload overlap then lead to a large communication overhead. Therefore, they propose to use a central message dispatcher, where applications can register their needs and the message dispatcher takes care of sending appropriate beacons.

5.2.7 Discussion

Advantages

The presented component achieves the basic security goals given in section 5.2.1. With the signatures, beacons are authenticated so that no impersonation is possible any more that can not be detected. In addition, beacons are automatically integrity protected as well. Regarding privacy, the component uses the identity and pseudonym management components, which ensure regular and thoughtful change of a vehicle's identity.

Drawbacks

One drawback of this basic solution is the problem that signature verification does not scale well. In scenarios with high node density, a node may receive more messages than it is able to verify in time. Therefore, further research needs to be done whether this situation can actually occur and if yes, which solutions could be applied. In addition, injection of bogus messages with high frequency may become a new attack vector, which then would have to be verified, which can lead to denial of service of the OBU due to the high load.

Besides the performance issue, some attacks are not addressed yet. For instance, dedicated shooting of single packets against single or all nodes can not be prevented nor detected by the current status. The same holds for handling jammed signals. In both cases, sort of a graceful degradation would be desirable.

Another issue is the strong dependency on the cryptographic components. Without these components on board, the implementation will not work.

5.3 Secure Flooding Component

5.3.1 Purpose of component

Assumptions on communication model:

Flooding is an approach that is used for a number of applications in VANETs to distribute information very quickly among the immediate surroundings of a vehicle. The basic principle involves multi-hop broadcast forwarding, which means that every node rebroadcasts the message once. As this can not be done network-wide, the rebroadcast is usually restricted by either a time-to-live counter value (TTL) or a geographic destination area (GDA).

Security Goals:

The purpose of this security component is to ensure integrity, authenticity and reliability of this mechanism. As a primary goal, the component is intended to prevent malicious vehicles being able to disturb the mechanism by means of rerouting, tampering and dropping. As a secondary goal, the module should be able to cope with attacks that intend to exploit the flooding mechanism to disturb the whole network operativeness. This is particularly important since flooding is a relatively costly mechanism that consumes a lot of bandwidth especially when node density is high.

5.3.2 Prerequisites

The component requires access to several services from other components. In particular, it relies on an established identity and credentials management, which allow for getting current credentials or even to have basic operations done there. For instance, the component does not care about the (current) identifier of a vehicle, its credentials or implemented cryptographic functions – it just uses the data and functions supplied in the corresponding components (see Section 3).

If pseudonyms are in use as described in the Privacy Module (see Section 4), the way of signing and verifying messages does not change - the privacy and identity management modules have to take care which keys to use.

The following list summarizes the dependencies:

- A mechanism to determine current identity and credentials
- An interface to sign data
- An interface to verify signed data
- An interface to get own location

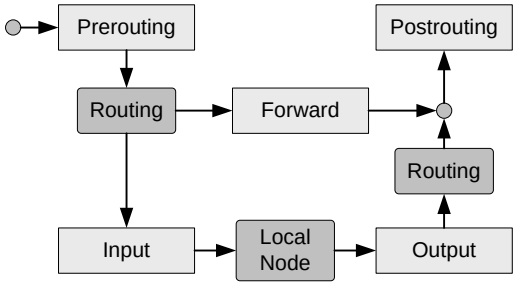

Figure 5.1: Details on hooking interception methods into the packet flow of the network stack

- An interface to get a global, loosely synchronized time

5.3.3 Interfaces and Services

Hooking

The component provides an interface which allows the component to be hooked into the process of flooding. Therefore, the component is notified when packets are to be sent, received, or forwarded.

The secure flooding component requires total control over the further processing of such packets. For example, the secure flooding component may decide whether it is secure to forward packets or if they are manipulated and should be discarded. This can be done either by this component itself, or by the network layer implementation. The interface design here assumes that the network layer is able to process certain return codes and has to drop a packet accordingly, if the secure flooding components indicates so.

Because we distinguish between packets being received, forwarded and sent, we assume that the hooks can be inserted at defined points of the network stack. This could be similar to the Linux netfilter architecture, as depicted in Figure 5.1.

For example, the `receive` method is added to the prerouting chain, the `forward` method to the forwarding chain and the `send` method to the output chain.

5.3.4 Description

Different actions need to be taken depending on whether a packet is incoming or outgoing, and in this case if it is created by the current node or forwarded only. Moreover, the applied security mechanisms partly depend on the mechanism used, i.e. whether the flooding restriction is TTL-based or GDA-based.

Outgoing messages

An outgoing message may either originate from the current node or is to be forwarded by the current node. The required security processing differs notably.

Own Messages:

For all messages created by one of the applications of node X, a signature has to be computed and a time stamp t_c has to be added if not already included (see discussion of overlapping fields of protocol and security mechanisms in Section 5.2.4). If the forwarding is TTL-restricted, then also a hash chain mechanism has to be applied, because otherwise malicious forwarders could decrease the TTL and thus increase the multi-hop propagation area. Such an increase leads of course to wasted network bandwidth. If the restriction is given by a fixed geographic destination region, the hash chain is not necessary.

Hence, the first step is to include a timestamp t_c or to ensure that an accurate timestamp is already included. This is done together with the signature.

The second step is to compute the hash chain in case of TTL-restricted fowarding. Therefore, the component has to generate a random base value v, apply a hash function TTL_{MAX} times on it and append the result h_v as well as v to the message.

As third step, the signature has to be created and the certificate for the used key (long-term ID or pseudonym) has to be attached. For this step, it is important to distinguish between mutable and immutable fields (F_m and F_{im}). Fields like the TTL value or the hash chain base value v change during the forwarding, whereas other, immutable fields like the payload, the source address or the end of the hash chain h_v do not change. The signature should only be computed for these immutable fields, and not include mutable ones.

For GDA-restricted forwarding, the message looks like this:

$$F_{im} = PAYLOAD \mid X \mid GDA \mid t_c$$

$$PACKET_{GDA} = F_{im} \mid sig_{SK_X}(F_{im}) \mid cert_{PK_X}$$

For TTL-restricted forwarding, the message includes the following:

$$F_{im} = PAYLOAD \mid X \mid h_v \mid t_c$$

$$F_m = v \mid TTL$$

$$PACKET_{TTL} = F_{im} \mid F_{me} \mid sig_{SK_X}(F_{im}) \mid cert_{PK_X}$$

Forwarded Messages:

Packets forwarded by the local node need to be processed after the routing procedure. In particular, the hash chain base value v has to be replaced by $h(v)$, i.e. the hash chain has to be shortened by one element, because the routing has decreased the TTL value.

Other fields, especially the signature and the immutable fields are not modified by forwarding nodes, but of course play a role to check incoming messages.

Incoming messages

The primary purpose for the inspection of all incoming packets is checking security policies. One of these policies is the verification of the attached signature as well as the certificate. If the signature or the certificate can not be verified, the message should be dropped. Moreover, more checks are necessary to ensure security. In summary, an incoming message should pass all the following checks before continuing processing (e.g. routing).

Certificate check:

The attached certificate is checked. The result can be stored for the lifetime of the certificate, so that further crypto operations for the same certificate are not necessary any more and the check can be skipped for further messages from the same source. The actual certificate checking is done by the identification management component.

Signature check:

The attached signature is verified. This task is provided by the identification management component.

Timestamp check:

To prevent replay, the timestamps of incoming messages have to be checked for consistency with the own time base. The own time base is provided in a secure manner by the hardware security module. The comparison should accept a configurable difference Δt between the packet's timestamp t_p and the local current time t_c.

GDA size check:

In order to prevent destination areas of very large size, a maximum covered surface as well as maximum width and length may be imposed on the GDA.

Hash chain check:

The hash chain used to secure the TTL decrement must be checked as well. Therefore, the node takes the base value v and applies the hash function sequentially TTL times on it. If the gained result matches the h_v value contained in the packet, the check is successful.

If any of these checks fails, the message must not be forwarded. Regarding local reception, it is either discarded immediately or marked as invalid by the component. The choice depends on whether applications also want to process invalid packets and should be configurable.

Secure Flooding API

Data structures

The packet structure is mainly defined by the communication system itself. However, the secure flooding requires some basic information in the packet. Moreover, certain data types like the node identifier are only placeholders for their actual implementation in the stack.

```
Packet {
  Identifier source
  Location   senderLocation
  int        floodingType
  <Optional Headers>
  <SecurityHeader>
  Data       data

  case(floodingType)
    TTL:
      byte   ttl
    GDA:
      GDA    destination
}

SecurityHeader {
  Signature   sig_SKX
  Certificate cert_PKX
  Time        t_c
  case(floodingType)
    TTL:
      byte[] v
```

```
      byte[] h_v
}
```

The following functions are hooked into the communication flow:

send

```
inputs:
    Packet p
outputs:
    SecurityHeader* secHeader;
    uint ReturnCode;
exceptions:
    InvalidPacket
```

Invoked on sending of a flood message which is created by the own node. Time, signature, and a certificate is always added to the security header. In case of a TTL restricted flooding, also the values h_v and v for the hash chain are appended to the security header. Note that the signature must comprise only immutable fields. The security header is returned to the calling communication stack, which has to append the header to the packet before sending.

receive

```
inputs:
    Packet p
outputs:
    uint ReturnCode;
exceptions:
    InvalidPacket
```

Checks if a recently received flood message is correct. Returns 0 (or nothing) if the packet was correct or an appropriate error code that may indicate a certificate error, signature error, timestamp error, GDA error, TTL error. Note that the input packet includes all received data, which also includes a SecurityHeader if the sender has successfully signed the packet.

forward

```
inputs:
    Packet p
outputs:
```

```
    SecurityHeader secHeader;
    uint ReturnCode;
exceptions:
    InvalidPacket
```

Forwards a recently received flood message. In case of TTL-based flooding, the new hash value v is calculated. The method returns a new SecurityHeader which replaces the old one, even though signature and certificate are not changed.

5.3.5 Performance

Again, one of the potential performance issues derives from the fact that potentially large amounts of signatures have to be verified, which has an impact on the processor if many verifications per time unit have to be done (see performance discussion of secure beaconing in Section 5.2.5).

5.3.6 Discussion

Advantages/Achievements:

With the security mechanisms applied, we achieve that only nodes with valid credentials (i.e. "'insiders"' with valid, certified pseudonyms) are accepted and only messages originating from such nodes are forwarded. Moreover, the applied signature also ensures the integrity of message content in transit. Time stamps help to prevent replay attacks which could otherwise be carried out also by outsiders without valid credentials.

Disadvantages/Drawbacks:

Both getting the own location correctly and assuming global time synchronisation are not trivial requirements. Of course, retrieving location and time is not within the purpose of the secure flooding component - e.g. a secure time base is to be provided by the hardware security module. But still, the secure flooding relies on it, and thus also has to note this as a potential drawback.

In addition, certain additional attacks of insiders and outsiders are not yet addressed. For instance, an attacker could try to disturb network availability by injecting messages at high frequency or he could jam the signal and still achieve more than only local impact.

5.4 Secure Routing Component

5.4.1 Purpose of component

Assumptions on communication model

While various applications in vehicular communication utilize broadcast mechanisms by nature, there are also other ones that need a unicast connection like vehicle-to-vehicle chatting. For other applications, the network layer first needs to transport messages to a certain area using single-path, hop-by-hop forwarding, and then distribute the message e.g. in a certain destination region.

Because vehicular networks have to deal with extremely mobile node, researchers have shown that topology-based routing mechanisms are not well suited for this network type. In contrast, position-based routing performs much better, since it does not require setup of end-to-end routes and is better suited to the application domain, where locations play a vital role.

Security Goals

The security component dealing with routing should assure the following goals:

- The sender of a message is a valid participant of the network
- Messages must not be manipulated by any forwarding node
- Messages should not be rerouted in the network

5.4.2 Prerequisites

The component requires access to several services from other components. In particular, it relies on an established identity and credentials management, which allow for getting current credentials or even to have basic operations done there. For instance, the component does not care about the (current) identifier of a vehicle, its credentials or implemented cryptographic functions – it just uses the data and functions supplied in the corresponding components (see Section 3).

If pseudonyms are in use as described in the Privacy Module (see Section 4), the way of signing and verifying messages does not change - the privacy and identity management modules have to take care which keys to use.

As we assume a geographic routing protocol, the component also needs a means to retrieve the node's own position for reference purposes. It depends on the setup of the system, where the position is drawn from. Without special accuracy or security considerations, this may be the on-board GPS.

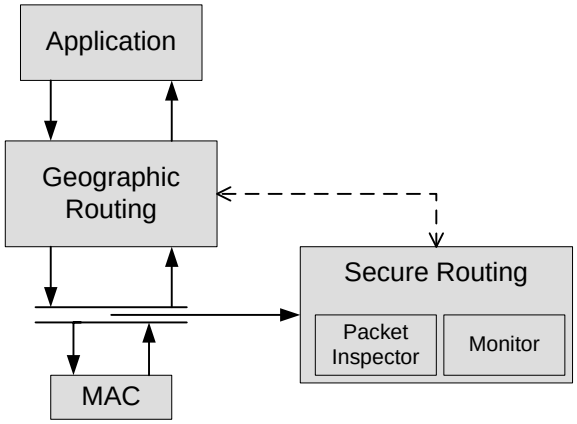

Figure 5.2: Secure Routing hook and control interface

The following list summarizes the dependencies:

- A mechanism to determine the key to use
- An interface to sign data
- An interface to verify signed data
- An interface to get own location
- An interface to get a global, loosely synchronized time

5.4.3 Interfaces and Services

The component provides an interface which allows the component to be hooked into the process of routing. Therefore, the component is notified when packets are to be sent, received, or forwarded.

The messages need to be truly handed over to the secure routing component, since it requires total control over the further processing of such packets. For example, the secure routing component may decide whether it is secure to forward packets or if they are manipulated and should be discarded.

Therefore, the component offers a notification and control interface for interaction with the routing itself. The routing consults the component at defined occasions during the routing process. For example, the decision to forward a message to a certain neighbor must be checked by the secure routing component to avoid forwarding packets to a malicious neighbor.

The potential interfaces are depicted in Figure 5.2.

5.4.4 Description

For the operation of the secure routing component, there are a number of specific states that have to be distinguished. For example, an application of the current node may send a message, and the router will forward it accordingly. In this case, specific actions have to be conducted, e.g. attaching signature and certificate, to provide evidence for other nodes that the sender is valid. In contrast, packets in transit that have to be forwarded must be verified. Moreover, more sophisticated attacks by corrupted insiders should be detected as well.

At the source of a message:

When a message is issued by an application that needs to be routed to a destination, the secure routing component adds a certificate and a signature over all immutable fields.

At an intermediate node:

To inhibit forwarding of invalid messages, the secure routing component at intermediate nodes has to verify the signature of a packet, before routing is performed.

At the destination node:

To prevent reception of bogus messages, the destination node has to verify the signature of a packet, before the packet is accepted.

If any of these checks fails, the message must not be forwarded. Regarding local reception, it is either discarded immediately or marked as invalid by the component. The choice depends on whether applications also want to process invalid packets and should be configurable.

Secure Routing API

Data structures

Like for every secure communication component, the packet structure is mainly defined by the communication system itself. However, the secure routing requires some basic information in the packet. Moreover, certain data types like the node identifier are only placeholders for their actual implementation in the stack.

```
Packet {
  Identifier source
  Location   destination
  <Optional Headers>
  <SecurityHeader>

  Data       data
}

SecurityHeader {
  Signature   sig_SKX
  Certificate cert_PKX
  Time        t_c
}
```

The following functions are hooked into the communication flow:

send

```
    inputs:
        Packet p
        Identifier designatedNextHop
    outputs:
        SecurityHeader secHeader;
        uint ReturnCode;
    exceptions:
        InvalidPacket
```

Sends a routed message which is created by the own node. Time, signature, and a certificate is added (by use of ID Management API). Note that the signature must comprise only immutable fields. The `designatedNextHop` is the node to which routing would forward the message. If this node is not trustworthy according to the findings of the secure routing module, the ReturnCode indicates that the packet can not be sent to the given next hop node.

receive

```
    inputs:
        Packet p
    outputs:
        uint ReturnCode;
    exceptions:
        InvalidPacket
```

Checks if a recently received message is correct. Returns 0 (or nothing) if the packet was correct or an appropriate error code that may indicate a certificate error, signature error, timestamp error. Note that the Packet includes the SecurityHeader in this case.

forward

```
inputs:
    Packet p
    Identifier designatedNextHop
outputs:
    SecurityHeader secHeader;
    uint ReturnCode;
exceptions:
    InvalidPacket
```

Forwards a recently received message. The method returns a new SecurityHeader which replaces the old one, even though signature and certificate are not changed.

isNeighborTrusted

```
inputs:
    Identifier node
outputs:
    boolean trusted;
exceptions:
    NeighborUnknown
```

This method allows to check whether the secure routing currently sees the node in question as trusted or not. The routing must use this function to determine whether a potential forwarder is trusted or not. The forward and send methods may decide not agree (which is to be configurable), if the routing decides to define a next hop that is not trustworthy.

5.4.5 Performance

Whenever asymmetric crypto operations are involved, an attack may be launched by massive injection of invalid packets. In this case, the sender can provoke a denial of service within his local radio range.

5.4.6 Discussion

By adding basic authentication and integrity mechanisms, potential attacks can be reduced, particularly by excluding illegal participants from the multi-hop communication.

However, many insider attacks are not covered by the current status. Due to billions of vehicles on the roads, the potential for malicious insider attacks is not neglectible. Therefore, additional mechanisms must be introduced in the future.

6 In-car Security Module

6.1 Overview

The in-car security module protects the interface of the wireless communication system to the in-car networks. Besides controlling the access to vehicle sensor data and services it also comprises mechanisms to ensure a correct provision of the data and services to the V2V/V2I applications.

The in-car security module provides:

- A firewall to control the flow of data and the access to services: who has access to what function/service on which in-car bus-system and what in-car function/service is allowed to send data out of the vehicle.
- An intrusion detection system (IDS) to monitor and check the status of the in-car networks to detect attacks via the communication system (e.g.; unusual frequency of requests, wrong order of requests, etc.) and to detect attacks via the in-car system (in this case, the data provided by the vehicle is not correct).

Based on the status of the in-car networks the IDS is able to dynamically adapt the configuration of the firewall: This adaptation ranges from denying access for selected applications to blocking the entire access to the in-car networks.

The structure of the in-car security module is depicted in figure 6.1.

The in-car security module consits of two components: firewall and IDS. Both components provide specific APIs (firewall API, IDS API). The in-car communciation API is an abstraction of the parts of the component APIs which are used by the V2V/V2I applications. The purpose of the adaption layer is the integration of the in-car security module into the vehicle network via the AUTOSAR standards.

Given the performance contraints in todays vehicle ECUs, the firewall component is designed as a network layer firewall (no application data is checked). The same holds for the IDS, by default, only header information is evaluated.

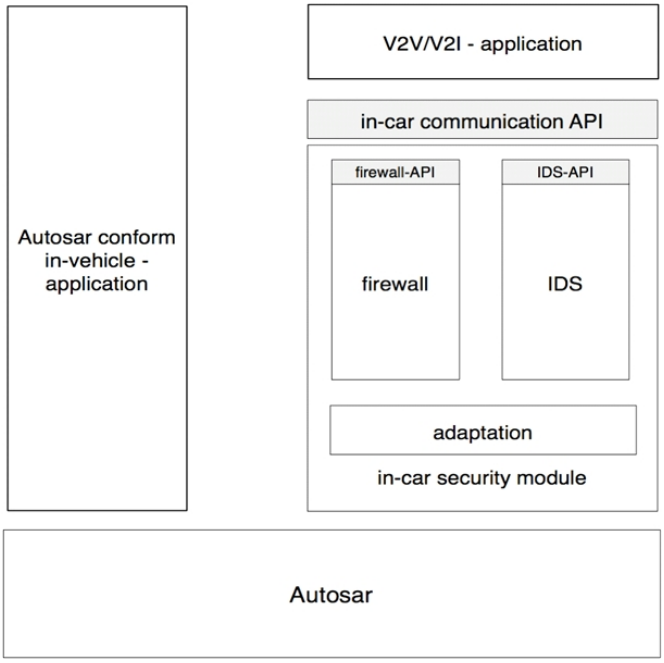

Figure 6.1: In-car security module

6.1.1 Interfaces and Services

The in-car security module has the following interfaces for access which comprise the connection process and the data exchange phase:

```
Session_ID incar_connect ( App_ID, App_Timestamp, Sensor_ID )
```

This function registers an application with the in-car security module to access the specified sensor. The application uses its ID for identification and transmits a timestamp for an additional possibility to check future communication. Sensors can be sources or sinks of information, e.g. a GPS positioning sensor or a display device for warning messages. Thus, the type of data which is transmitted depends on the selected sensor.

The return value is a Session_ID, which is used to identify the application in future communication. A valid Session_ID has a value greater than 0; if the application is not granted access a Session_ID ≤ 0 is returned.

For each tuple of App_ID and Sensor_ID a separate Session_ID is assigned, which is stored in an internal session table of the in-car security module.

```
Bool incar_disconnect ( Session_ID, App_ID, App_Timestamp, Sensor_ID )
```

This function deregisters the access of an application for a specific sensor. This sensor is checked out from the module and the tuple is removed from the session table.

The application uses its App_ID and the Session_ID for identification and transmits the initial timestamp for an additional verification.

The return value is true if Session_ID, App_ID and App_Timestamp match with the internal session table of the in-car security module, false else wise. Deregistration takes place in any case.

```
Data incar_data_retrieve( Session_ID )
```

This function retrieves the current data value, e.g. the current GPS position, for a specified Session_ID, based on a pull concept. The Session_ID comprises the selection of the sensor and its data, as described before.

Prerequisite is a successful previous registration and a valid Session_ID.

```
Void incar_data_transmit( Session_ID, App_data )
```

This function transmits the App_data value for the specified Session_ID, e.g. a warning message.

Prerequisite is a successful previous registration and a valid Session_ID. The App_data is checked by the in-car security module, e.g. for correctness of the format and specifications (see Sec. 6.3).

6.2 In-Car Security Firewall Component

6.2.1 Purpose of component

The firewall component of the in-car security module is designed to protect the in-car network against attacks from the wireless telematic/C2XC interface. For this reasion a packet filter will be implemented based on the source-, destination adress and if available the destination service/port. The firewall packet rule type table shall be updated by the component Intrusion Detection System. The Intrusion Detection System can add new firewall rule type table entries to deny specific target or destination adresses. This includes also that specific services are denied.

6.2.2 Prerequisites

The firewall required different types of automotive communication and protocol stacks, as in general the in-car security Module. A running autosar will fulfill this requirements.

6.2.3 Interfaces and Services

The firewall component offers in general the following interfaces and services:

- API to send, receive and register messages via the in-car security module api
- API for the update of the packet rule type table. The access is limited to the Intrusion Detection System

6.2.4 Description

In general a firewall is a device which separates two or more networks with different security levels by means of a packet rule type table. In our case it is the wireless telematic interface and the in-car-networks. The telematic network can be seen as the insecure one and the in-car networks are the secure ones.

The SeVeCom firewall will be a packet based firewall. For this reason it will inspect the header data of each message that targets the in-car network. According to the internal packet rule type table a message can pass the firewall or will be denied. The firewall denies all traffic between the wireless telematic interface and the various in-car networks by default, e.g. FlexRay, CAN, LIN or MOST.

The packet rule type based table must be known for all possible configurations during the compile time. The table should be static due to reduction of complexity from the telematic application point of view. The packet rule type table can be updated based on the IDS internal status. Via a set of api functions the IDS can deny dedicated packets or block the whole traffic via the wireless interface.

The telematic application must register to the firewall via the in-car module if the application wants to send or receive messages to/from the in-car network. Due to the pre configured packet rule type table in the denied state the firewall will set the inactive rule to the active state and allows this kind of traffic. This allows a later security enhancement with signature based application registration. After this step the telematic application can send or receive messages from the in-car network.

6.2.5 Performance

Today the required performace of the firewall ist not so clear. From US projects the number of 5000 signature proofs per second is known. What this implies for the needed performance of the in-car firewall can only be estimated. We assume that 10% of the signature proofs end in a communcation traffic target to the in-car networks at maximum.

6.2.6 Related Work

The firewall component is integrated into the vehicle network via the AUTOSAR standards (www.autosar.org). Some aspects of telematic gateway are also discussed in the EASIS project, see www.easis.org.

6.2.7 Discussion

To enhanced the security of the in-car network the firewall technique used in IT based systems will be used. The processing power and resources avalaible for the firewall is in general limited for embedded systems. A so-called Application-Layer Firewall will not be considered, because the telematic application will take over this duty, see D3.1 for details. A possible add-on could be a signature based registration for accessing the in-car network via the firewall.

6.3 In-Car Security Module Intrusion Detection System

6.3.1 Purpose of component

The Intrusion Detection System (IDS) has the purpose to automatically detect unwanted manipulations of data. Hence, in addition to preventive measures, the IDS is supposed to provide real-time protection and automatic attack detection of ongoing attacks and manipulation attempts. In a second step the IDS can be extended towards an Intrusion Prevention System (IPS), which does not only allow the detection of attacks but also supports measures of reaction in order to stop an attack once it has been detected, or mitigate the negative effects for the system.

6.3.2 Prerequisites

In general, a basis to detect the possible attacks has to be defined adequately in advance. Depending on the architecture and techniques that are implemented by the IDS, different prerequisites apply. Mainly the knowledge base of the IDS needs to be available for the IDS to ensure proper operation. This knowledge has to be included into the system before operation.

6.3.3 Interfaces and Services

The IDS is connected to the packet rule type table of the firewall, in order to enable updates of the configuration in case of an attack or detected anomaly.

6.3.4 Description

The IDS will contain an anomaly based approach: Anomaly Based Systems contain knowledge about the normal behaviour of the system. The system continuously monitors the behaviour and generates an alert, once an action which is unusual and not part of the normal behaviour is recognized. Anomaly Based Systems can either rely on knowledge which is acquired within a learning phase and further on used as normal behaviour, or the normal behaviour of the system is specified manually as an input for the IDS. The latter solution requires more effort for definition and specification of the system but provides a lower rate of false positives.

The IDS has access to the configuration of the firewall. Especially, it is allowed to change the configuration adequately once the status of the IDS requires it. This can be the case if an intrusion is detected.

Moreover, the IDS has access to different information sources inside the vehicle. These information sources will observe the functionality of dedicated components of the vehicle and monitor the activities during operation. For the analysis of the data which is monitored, two different approaches are considered:

1. All data collected by the information sources is transferred to the IDS which is responsible for further processing of the data. This keeps the information sources fairly simple and cheap. All intelligence for the intrusion detection is contained in the IDS.
2. Before transferring data to the IDS the information sources themselves perform some analysis and pre-processing of the data. Therefore each information source is equipped with knowledge about the component which is observed. This knowledge allows to perform intelligent detection of attacks on the information source itself.

For the detection of intrusions the IDS or the information sources will have access to an IDS information base which contains necessary data like configurations, etc. for the system to decide upon. If the data provided by IDS information sources is not consistent with the data derived from the information base, an alert is raised. To raise an alert, the IDS establishes a connection to the firewall configuration and performs a change of the configuration which adequately reflects the detected inconsistency. If necessary, current connections to the vehicle which are permitted by the firewall can be de deactivated. This allows the IDS to react when an intrusion has been detected. Based on this system model, the detection is performed and afterwards the event is serialized and prepared for logging. The detection is implemented by the IDS through the analysis of data packet headers. However, an extension for the additional analysis of data contents is possible with respect to the performance situation of the system.

6.3.5 Performance

The system needs to be able to perform all information-collection from the sensors, processing of the data and the detection of attacks in real-time. Especially for the first approach the load on the bus system is fairly high. This has to be taken into account for the development of the final deliverables.

6.3.6 Related Work

The in-car security module is integrated into the vehicle network via the AUTOSAR standards (www.autosar.org).

6.3.7 Discussion

The IDS improves the security status of the in-vehicle architecture. In general the inclusion of an IDS into today's vehicle architectures requires the availability of sufficient processing power and resources for the system. As described above these resources can be distributed over the system or exclusively located in the IDS itself.

7 Crypto Support Module

7.1 Overview

The general purpose of the Crypto Support Module is to provide the implementation of the cryptographic operations needed by the applications running on the OBU. The applications can call these cryptographic operations through the APIs provided by the components of the Crypto Support Module. As illustrated in Figure 7.1, there are two components in this module: the OBU Crypto Component and the HSM Component. The OBU Crypto Component is a software component running on the OBU that implements the public key cryptographic operations, such as digital signature verification, and provides a wrapper for the private key operations. The private key cryptographic operations themselves, such as digital signature generation, are implemented in the HSM Component, which is an independent tamper-resistant device connected to the OBU. Management applications can directly call some functions of the HSM API to perform management operations, such as initializing the device, revoking its root public keys, and revoking the device itself. Sections 7.2 and 7.3 give more details about the OBU Crypto Component and the HSM Component, respectively.

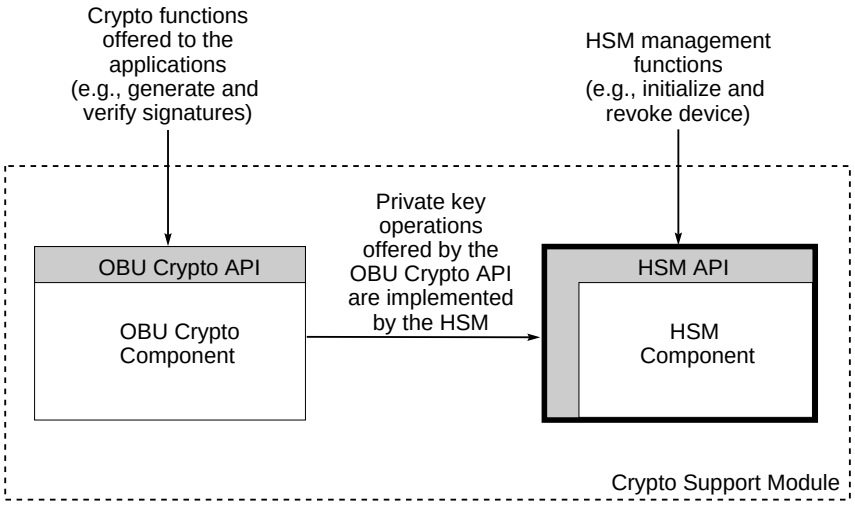

Figure 7.1: Components and interfaces of the Crypto Support Module

7.2 OBU Crypto Component

7.2.1 Purpose of component and prerequisites

The purpose of the OBU Crypto Support Component is to provide a general interface to cryptographic functions for the applications, and to implement those cryptographic functions that use only public information (i.e., public keys). The cryptographic functions that use private information (i.e., private keys) are implemented in another component, called HSM Component that, in the ideal case, is realized as a separate hardware device that also provides physical protection measures to safe-guard the sensitive private information. Thus, the OBU Crypto Component relies on the HSM Component and uses the services provided by the HSM Component through the HSM API.

The OBU Crypto Component can be implemented as a software module running on the OBU. This may be preferable to integrating it with the HSM Component for performance reasons, as the OBU may have more computing resources than the typical devices that can function as a HSM. Note also that the vehicle needs to verify digital signatures, which is a public key operation, with a potentially very high rate. Therefore, it makes sense to implement the public key cryptographic operations in the OBU rather than in the HSM.

7.2.2 Interfaces and Services

The OBU Crypto Support Component provides cryptographic services, including *digital signature generation and verification* and *public key encryption and decryption*, as well as *hash value computation* through the following API:

authenticateData

```
inputs:
    uint KeyID;
    byte* Data;
    uint DataLength;
    byte TimeStampLength;
outputs:
    uint64 Timestamp;
    byte* SignedHash;
    uint SignedHashLength;
exceptions:
    UnknownKeyID;
    WrongKeyType;
```

```
    WrongTimeStampLength
```

Timestamps and digitally signs the given input data with a given credential as specified in the HSM Component description. This is a wrapper function, the timestamp and the signature are generated in the HSM Component(for a more detailed description, see the signWithShortTimestamp function in the HSM Component API). The possible values of TimeStampLength are 0 and 1 (shortTimeStamp, longTimeStamp).

verifyAuthenticatedData

```
inputs:
    privateKey VerificationKey;
    byte* Data;
    uint DataLength;
outputs:
    byte ReturnCode
exceptions:
    UnknownKeyID;
    WrongKeyType;
```

Verifies the authenticity of the given authenticated data by verifying the digital signature on the data. The data may be timestamped or not. If the verification is successful, it returns a 0 ReturnCode.

encryptPlainTextData

```
inputs:
    privateKey EncryptionKey;
    byte* Data;
    uint DataLength;
outputs:
    byte* EncryptedData
    uint EncryptedDataLength
exceptions:
    WrongKeyType;
```

Protects for confidentiality of (encrypts) the given data, and returns the encrypted data.

decryptEncryptedData

```
inputs:
    uint KeyID;
    byte* EncryptedData;
    uint EncryptedDataLength;
outputs:
    byte* DecryptedData;
    uint DecryptedDataLength;
exceptions:
    UnknownKeyID;
    WrongKeyType;
    DecryptionFailure;
```

Decrypts (obtains plain text data from) encrypted data. This is a wrapper function, the decryption is performed by the HSM Component(for a more detailed description, see the decrypt function in the HSM Component API).

generateRandomData

```
inputs:
    uint NumberOfBytes;
outputs:
    byte* RandomBytes;
    uint RandomBytesLength;
exceptions:
    --
```

Generates random data. This function is a wrapper function to the HSM Component (for a more detailed description, see the getRandom function in the HSM Component API).

generateHashValue

```
inputs:
    byte* DataToBeHashed;
    uint DataToBeHashedLength;
outputs:
    byte* HashValue;
exceptions:
    --
```

Generates the one-way hash of the input data.

7.3 HSM Component

7.3.1 Purpose of component

Implementing security services for vehicular communications requires the vehicles to store sensitive data [MRJPH05], such as cryptographic keys, event logs, etc. It must be assumed that potentially malicious parties, such as maintenance service providers or even the vehicle owner, can have unsupervised access to the vehicle for extended periods of time. In addition, these potentially malicious parties may have incentives to compromise the sensitive data stored by the vehicle. For these reasons, the sensitive data needs to be protected from unauthorized access by physical means.

In SeVeCom, we envision that the vehicles are equipped with a Hardware Security Module (HSM). The purpose of the HSM is *to store sensitive information* within the vehicle and *to provide physical protection measures* to safeguard sensitive information. This mainly means the storage and the physical protection of sensitive cryptographic keys (e.g., private keys for signature generation). In addition, the HSM must be able *to perform cryptographic operations* (e.g., generate digital signatures) with the stored keys in order to ensure that sensitive information never needs to leave the physically secured environment provided by the HSM.

At a high level, the HSM serves as the basis of trust in the SeVeCom security architecture. In particular, without the physical protection provided by the HSM, the signature generation keys could be easily compromised, and then used to generate fake messages that appear to be authentic. Hence, in that case, the vehicles could not trust even the signed messages, and therefore, the entire security architecture would be more or less useless.

7.3.2 Prerequisites

The HSM is the heart of the SeVeCom security architecture. It does not rely on other components, but rather other components, such us the secure beaconing, communications, and routing protocols use the HSM, mainly for digital signature generation purposes.

The HSM must satisfy some timing requirements that are determined by the applications that use it. The most stringent timing requirements are determined by the periodic beaconing. Periodic beaconing means that the vehicle periodically broadcasts its position, speed, and direction of movement, and in this way, it informs nearby vehicles about its presence. Many vehicle safety applications (e.g., collision avoidance, lane merge assistant, etc.) rely on this mechanism. These periodic beacon messages need to be digitally signed, which means that typically, the HSM must be able to generate a few tens of digital signatures per second.

Note that the vehicle may be required to verify an order of magnitude more digital signatures when receiving beacons from nearby vehicles. However, signature verification is a computation that uses only public information (i.e., public keys), and therefore it can be performed outside of the HSM, typically, on the OBU of the vehicle.

7.3.3 Interfaces and Services

The main service provided by the HSM to the components that use it is the *generation of digital signatures*. In order to support this, the HSM also provides *key management* services. In particular, the HSM must be able to generate (or import) the private keys corresponding to the anonymous public keys of the vehicle. Furthermore, the HSM must also be able to process revocation commands originating from a trusted authority.

In addition to the digital signature generation service, the HSM performs *time stamping*. This means that the HSM is equipped with a real-time clock, and before signing a message, it inserts a timestamp in the message. The timestamp is output by the HSM together with the generated signature. Furthermore, the HSM can also perform *decryption of encrypted messages* with the private decryption keys that it stores.

Finally, the HSM may offer a *secure storage service* for logging purposes (e.g., to implement event data recording functions). Actually, the memory of the HSM may not be enough to store a large amount of data, and therefore, the data are stored in external storage in an encrypted form in such a way that only the HSM can decrypt the data. In this specification of the baseline architecture, we do not further specify the secure storage service of the HSM.

The HSM has some hardware interfaces and it has an Application Programming Interface (API). The hardware interfaces include an interface for power supply and an I/O interface through which the HSM can interact with other in-vehicle components, such as the OBU of the vehicle.

The API is the top level software interface of the HSM through which the services provided by the HSM can be reached by the components that use the HSM. In other words, the API provides the means to invoke the digital signature and time stamping service, the decryption service, and the key management service of the HSM. A detailed description of the API of the HSM is provided in Subsection 7.3.4.

7.3.4 Description

HSM architecture

The hardware architecture of the HSM is illustrated in Figure 7.2. The HSM has a CPU, a memory module, and some non-volatile storage. In addition, in order to ensure the freshness of the cryptographically protected messages produced by the HSM, it must also have a real-time clock, and consequently, a battery module that ensures the independent operation of that clock. Finally, the HSM also has a hardware random number generator that is used for key generation purposes.

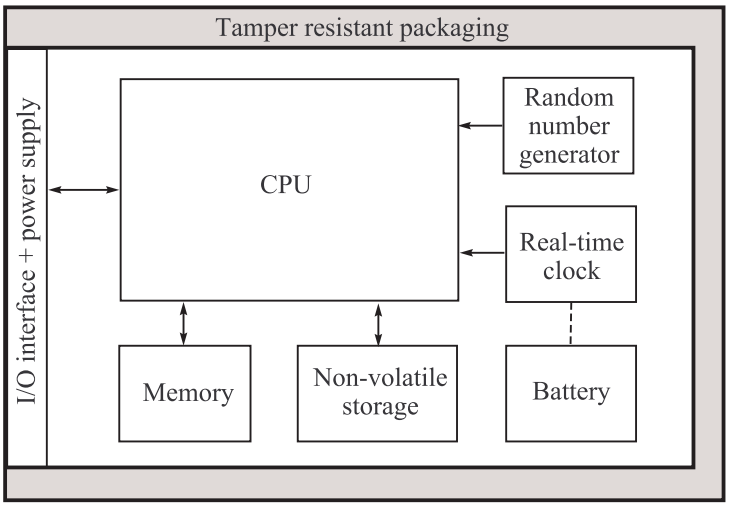

Figure 7.2: Architecture of the HSM

Level of physical protection

We require the HSM to be physically protected against tampering; indeed, this property of the HSM is where trust in it is derived from. The physical protection of the HSM should ensure at least tamper evidence. However, this is not enough, as regular inspection of the vehicles is rather infrequent (e.g., in some countries it happens in every second year), which results in a large vulnerability window. Therefore, we also require that the physical protection of the HSM also ensures some level of tamper resistance. We understand that high-end tamper resistant hardware modules are very expensive, therefore, in order for our baseline architecture to be practically feasible, we require only a minimal level of tamper resistance that can be achieved with special packaging and coatings.

Time stamping and digital signature generation

The main services provided by the HSM are the time stamping and the digital signature generation services. These services are always provided together. This means that every signature generated by the HSM contains a timestamp value. We require this joint provision of services because we do not want the HSM to be capable of interpreting the messages that it signs. In other words, if the HSM generated signatures without including its own timestamp, then it should be able to check timestamps that are inserted into the messages by the potentially insecure applications. However, different applications may use different message formats, and thus, the HSM must be able to interpret all those message formats. This may easily lead to interoperability problems. We avoid this by requiring that the HSM always inserts its own trusted timestamp value in the signatures that it generates.

When invoking the time stamping and digital signature generation service, the HSM is provided with the data to be signed and the key identifier of the private signature generation key to be used. The HSM inputs these data through its I/O interface. Using the received key identifier, the HSM retrieves the signature generation key and the corresponding hash algorithm, signature algorithm, and parameters from its internal memory. Then, it attaches the current timestamp to the data to be signed, computes the hash value of the time stamped data, and generates the digital signature on the hash value. Formally, the HSM computes

$$\sigma = sign_{K^{-1}}(h(data|T))$$

where K^{-1} is the private signature generation key, $data$ is the data to be signed, T is the current timestamp value, h is the hash function used, $sign$ is the signature generation function used, and $|$ denotes concatenation. The HSM outputs the resulting signature σ and the timestamp value T, which can be attached by the OBU to the message that is signed before sending that message.

As the HSM does not check the consistency and the validity of the data that it signs, it is not necessary to supply the entire message to be signed to the HSM. Instead, the HSM can receive only the hash value of the message. In that case, the HSM generates a digital signature on $data = h(message)$, but this is entirely transparent to the HSM. At the same time, applications that verify digital signatures must be aware of the fact that the input $data$ of the above signature computation was the hash value of the message.

Key management

The HSM maintains five types of keys: (i) short-term signature generation keys that are typically used to authenticate the short-term pseudonyms used by applications

running on the vehicle, (ii) short-term decryption keys that are used to decrypt encrypted messages intended to the applications running on the vehicle, (iii) a long-term signature generation key that is used to authenticate the real identity of the vehicle, (iv) a long-term decryption key that is used to decrypt encrypted messages intended to the vehicle itself, and (v) two long-term root public keys that are used to verify the authenticity of commands (e.g., revocation of the HSM) sent by the authorities to the HSM.

Note that as certificates do not contain private data, they are not stored inside the HSM. Instead, the certificates that correspond to the private keys stored in the HSM are maintained by a certificate management application in the OBU. For similar reasons, the public keys that correspond to the private keys are not stored in the HSM either.

The keys that are maintained by the HSM are stored in an internal key database. An entry in this database contains at least the following information:

- a key identifier;
- the value of the key;
- a reference to the cryptographic algorithm and associated parameters that should be used with the key of this entry;
- some flags indicating the type of the key (e.g., long-term root public key, long-term signature generation key, etc.), whether it is removable from the HSM or not (short-term keys are removable in general, whereas the long-term keys are not), and whether it is under update (long-term keys can be only updated, not removed)
- a lock counter that indicates the number of applications currently using the key of this entry.

Below, we first describe the management of the short-term keys, and then we briefly specify the management of the long-term keys.

Short-term signature generation keys.

Typically, these keys are intended to sign the periodic beacon messages broadcast by the vehicle. Thus, for privacy reasons, the public keys that correspond to these short-term private keys may be certified in an anonymous manner by a trusted third party, called the *pseudonym provider*. An anonymous certificate contains only the public key, the validity period of the certificate, the identifier of the issuer, and the digital signature of the issuer. In particular, it does not contain the identifier of the vehicle to which it has been issued. Note, however, that this issue is entirely transparent to the HSM, as it does not maintain certificates. This means that the logic that controls the pseudonym usage and the periodic change of pseudonyms is not part of the HSM; rather the HSM

only supports the pseudonym management application by generating short-term key pairs, storing the private keys, and computing signatures (when requested).

The HSM can be instructed (through its API) to generate a new short-term signature key pair. For this, one must specify the corresponding signature algorithm (e.g., ECDSA) and its parameters (e.g., the curve and the base point in case of ECDSA), as well as the hash algorithm that will be used for hashing the data before computing signatures. When the HSM generates a new key pair, it creates a new entry in the internal key database and it stores the private key together with the corresponding context information (type, algorithms, and parameters) and key identifier. The HSM outputs the public key and it is the responsibility of the external applications running on the OBU to obtain a certificate for it. However, certificate requests can be passed back to the HSM for being authenticated, typically with the long-term master signature generation key of the HSM. But again, this is transparent to the HSM, because certificate requests are treated as any other data to be signed by the HSM.

The HSM can be instructed (through its API) to increase the lock counter of a short-term private key. The HSM ensures that a key that has a lock counter greater than zero will not be deleted from the database. This feature can be useful when multiple applications use the same key concurrently. Note that locking is only meant to prevent that one application deletes a key that is still used by another application, and *not* to prevent the usage of the key for generating signatures. This means that a malicious application cannot prevent other applications from obtaining signatures from the HSM. At the same time, a malicious (or malfunctioning) application may increase the lock counter of a key and "forget" to decrease the counter when it finished using that key, which may result in a situation where the HSM cannot generate new keys as it has no more memory to store them. This leads to a breach in privacy (as pseudonyms cannot be refreshed) but not in security (as private keys still remain secret).

As the internal memory of the HSM is not infinite, we must also allow the deletion of keys from the key database of the HSM. Thus, the HSM can be instructed (through its API) to delete a short-term private key. The HSM then verifies the lock byte of the given key. If it is zero, then the HSM removes the entire entry corresponding to the given key from the key database. Otherwise, if the lock byte is greater than zero, the entry is not deleted.

Short-term decryption keys.

The management of these keys is very similar to the management of the short-term signature generation keys. The HSM can be instructed (through its API) to generate a new short-term encryption key pair. For this, one must specify the corresponding encryption algorithm (e.g., ECIES with HMAC-SHA1 and AES-CBC) and its parameters (e.g., the key length of AES and the elliptic curve domain parameters). When the HSM generates a new key pair, it creates a new entry in the internal key database and it

stores the private key together with the corresponding context information (type, algorithms, and parameters) and key identifier. The HSM outputs the public key and it is the responsibility of the external applications running on the OBU to obtain a certificate for it. Locking and deleting short-term decryption keys is done in the same way as for short-term signature generation keys.

The long-term signature generation key.

The long-term signature generation key of the HSM is used to authenticate the real identity of the HSM. This key is generated by the HSM before it begins its operation (typically at manufacturing time), and it cannot be deleted or revoked just exchanged. If it gets compromised, then the entire HSM must be revoked by a command that can be authenticated with one of the long-term root public keys stored in the HSM. The long-term signature generation key is exchanged if its certificate is to expire. The exchange is executed in two steps. In the first step, the Identity Management Module initiates the generation of a new long term signature generation/verification key pair. In the first phase, the HSM only stores the new keys, but does not use them. In the second step, the HSM deletes its old key, and starts to use the new key. The second step is started by a swap command containing the new long-term signature verification key, authenticated with one of the root keys.

The long-term decryption key.

The long-term decryption key of the HSM is used to decrypt encrypted messages that are intended to the vehicle. This key is generated by the HSM before it begins its operation (typically at manufacturing time), and it cannot be deleted or revoked just exchanged. If it gets compromised, then the entire HSM must be revoked by a command that can be authenticated with one of the long-term root public keys stored in the HSM. The exchange process of the long-term decryption key is the same as the long-term signature generation key's.

The long-term root public keys.

There are two long-term root public keys in the HSM that are used to authenticate commands sent by the authorities to the HSM. These keys are loaded into the HSM during its initialization phase in a secure environment. Once the initialization phase is completed, the HSM does not allow to load new root public keys in it. Still, it may be necessary to revoke or update a root public key in the HSM for various reasons (including the possibility of the corresponding private key being compromised). It would be very impractical to require that in case of a root key revocation or update, the entire HSM is replaced with a new one, because the same root public key may be used by a potentially large number of HSMs. Therefore, we want to allow the revocation and the update of root public keys through the API of the HSM.

As we mentioned above, it may be possible that a root public key is compromised (i.e., its corresponding private key is leaked out), however, we assume that the event when both root keys are compromised at the same time has an extremely small probability (e.g., if they stored at physically distinct locations). In other words, we assume that in practice, at most one root key is compromised at any time.

A root public key K can be revoked by sending a revocation message to the HSM (through its API) that is signed with the private key corresponding to K. When one of the root public keys, say K_1, of the HSM is revoked, *and only in this case*, the HSM accepts a new root public key K_1' if K_1' is signed and the signature can be successfully verified with the other root public key K_2 of the HSM.

The rationale behind this scheme is the following: If the private key corresponding to a root public key K_1 is compromised, it can only be used to revoke K_1. In particular, the other root key K_2 cannot be revoked with the compromised key K_1. In addition, if K_1 is compromised, then the other root key K_2 must be intact by assumption. Thus, it is safe to accept a new root key K_1' if it is signed and the signature can be verified with K_2.

HSM API

In this subsection, we specify the API of the HSM that offers the functions through which the above described time stamping, digital signature generation, decryption, and key management services can be invoked.

Algorithm identifiers, key identifiers, and flags

The HSM handles each supported cryptographic algorithm and its parameters as a single *bundle*. Applications can refer to a bundle with an algorithm identifier, which is a 16-bit unsigned integer. In this way, an algorithm identifier implicitly specifies the parameters of the algorithm (e.g., the length of the key to be used with that algorithm). Tables 7.2 and 7.3 list the supported algorithm and parameter bundles for signature and encryption, respectively, and their associated algorithm identifiers.

Applications can refer to the keys stored in the HSM by key identifiers, which are 16-bit unsigned integers generated by the HSM. Keys stored in the HSM have some flags associated with them that specify their type (e.g., long-term root public key) and status (removable or not). These flags are represented by the bits of a single byte as specified in Table 7.1. Moreover, keys have an associated lock counter as described above.

By definition, there are 2 long-term root public keys in the HSM, which always have the key identifiers `0x0001` and `0x0002`, respectively. In addition, the HSM has a long-term signature generation key and a long-term decryption key which have key identifiers `0x0003` and `0x0004`, respectively. The key identifiers in the range `0x0005-0x00FF` are

Bit no.	Meaning	Possible values
1-4	key type	`0x0` = long-term root public key `0x1` = long-term signature generation key `0x2` = long-term decryption key `0x3` = short-term signature generation key `0x4` = short-term decryption key
5	is_removable	`0` = no `1` = yes
6	under_update	`0` = no `1` = yes
7-8	n/a	n/a

Table 7.1: Specification of the bits in the flags byte associated to the keys stored in the HSM.

reserved for future use. The remaining key identifiers in the range `0x0100-0xFFFF` can be assigned to the short-term private keys generated and stored in the HSM.

In addition, the long-term keys are always non-removable (i.e., their `is_removable` flag is set to 0), while the short-term keys are removable by default. At the same time, the short-term keys can be locked and unlocked by the applications at will.

Data structures

We define only the following two data structures that we use in the specification of the API of the HSM:

```
struct {
        uint AlgorithmID;
        byte* PublicKeyValue;
} public_key;

struct {
        uint KeyID;
        byte Flags;
        uint LockCounter;
} key_info;
```

The `public_key` data structure contains a pointer to a public key value and an algorithm identifier that specifies the bundle of algorithms and parameters associated with the given key. Recall that the algorithm identifier implicitly specifies the length of the key,

that is why the `public_key` data structure does not contain an explicit `PublicKeyLength` field.

The `key_info` data structure contains a key identifier, they flags, and the lock counter value associated to a given key.

signWithShortTimestamp

```
inputs:
    uint KeyID;
    byte* Data;
    uint DataLength;
outputs:
    uint32 Timestamp;
    byte* SignedHash;
    uint SignedHashLength;
exceptions:
    UnknownKeyID;
    WrongKeyType;
```

This function inputs a key identifier and some data, and generates a digital signature on the data using the private key associated with the key identifier. As explained earlier, the HSM always generates signatures together with a timestamp, hence, besides the signature itself, this function also outputs the timestamp value that has been used in the signature generation process. Specifically, this function uses a short timestamp, which is a 32-bit unsigned integer that represents the *seconds* that elapsed since 0:00am January 1, 1970.

This function throws an exception if it is called with an unknown key identifier or with a key identifier that refers to a key that cannot be used for signature generation. This latter situation may occur when the key identifier refers to a long-term root public key or a decryption key.

signWithLongTimestamp

```
inputs:
    uint KeyID;
    byte* Data;
    uint DataLength;
outputs:
    uint64 Timestamp;
    byte* SignedHash;
    uint SignedHashLength;
exceptions:
```

```
    UnknownKeyID;
    WrongKeyType;
```

This function works in the same way as the `signWithShortTimestamp` function except that it uses a long timestamp value, which is a 64-bit unsigned integer that represents the *microseconds* that elapsed since 0:00am January 1, 1970.

decrypt

```
inputs:
    uint KeyID;
    byte* EncryptedData;
    uint EncryptedDataLength;
outputs:
    byte* DecryptedData;
    uint DecryptedDataLength;
exceptions:
    UnknownKeyID;
    WrongKeyType;
    DecryptionFailure;
```

This function is used to decrypt encrypted messages. It inputs the key identifier of the decryption key to be used and the encrypted data itself. Upon successful completion, the decrypted data is returned.

This function throws an exception if an unknown key identifier is supplied, or the key identifier refers to a key that cannot be used for decryption. In addition, an exception is thrown if the decryption operation fails due to wrong formatting or invalid message authentication (depending on the type of encryption used).

generateKeyPair

```
inputs:
    uint AlgorithmID;
outputs:
    uint KeyID;
    public_key PublicKey;
exceptions:
    UnknownAlgorithmID;
    NotEnoughMemory;
```

This function can be used to generate a new short-term key pair in the HSM. The application must specify an algorithm identifier, which specifies the cryptographic algorithm

and implicitly specifies the parameters of that algorithm that are needed for the key pair generation. Once the new key pair is generated, the HSM also generates and assigns a new key identifier to it. Then the HSM stores the private key in its internal memory, and outputs the key identifier and the public key.

This function throws an exception if the supplied algorithm identifier is not known or there is not enough free memory in the HSM to store the key to be generated.

removeKey

```
inputs:
    uint KeyID;
outputs:
    --
exceptions:
    UnknownKeyID;
    KeyIsNotRemovable;
    KeyIsLocked;
```

This function allows an application to remove a key from the internal memory of the HSM, and in this way, to create enough free space to store newly generated keys. The caller must supply the key identifier of the key to be removed. Upon successful completion of this call, the HSM deletes the given key and returns nothing.

This call throws an exception if the supplied key identifier is unknown, or the key identifier refers to a key that is not removable from the HSM, or the key identifier refers to a key that is removable but currently locked in the memory by some applications.

increaseLockCounter

```
inputs:
    uint KeyID;
outputs:
    --
exceptions:
    UnknownKeyID;
```

This function allows an application to increase the lock counter of a key in the memory of the HSM. Locking means that the HSM ensures that a key with a lock counter greater than zero will not be removed from the device. It is also ensured that the lock counter cannot overflow, meaning that it has a maximum value and it cannot be increased beyond that maximum. The input to this function is the key identifier of the key and upon successful completion of the call nothing is returned.

This function throws an exception if the supplied key identifier is unknown.

decreaseLockCounter

```
inputs:
    uint KeyID;
outputs:
    --
exceptions:
    UnknownKeyID;
```

This function allows an application to decrease the lock counter of a key. It is ensured that the lock counter cannot become smaller than zero. This function inputs the key identifier of the key, and upon successful completion of the call nothing is returned.

If the supplied key identifier is unknown, then an exception is thrown.

getKeyInfo

```
inputs:
    uint KeyID;
outputs:
    key_info KeyInfo;
exceptions:
    UnknownKeyID;
```

This function can be used by applications to read the flags and lock status of a key out of the HSM. It returns a `key_info` data structure that contains the requested information.

If the supplied key identifier is not known, then an exception is thrown.

getKeyRingInfo

```
inputs:
    --
outputs:
    key_info* KeyRingInfo;
    uint KeyRingSize;
exceptions:
    --
```

This function allows an application to get information about all the keys stored in the HSM with a single call. It returns the number of keys (i.e., the size of the key ring) and a pointer to an array of `key_info` data structures that contain the information associated with the keys.

getChallengeForTimeSynch

```
inputs:
    --
outputs:
    byte[16] Challenge;
exceptions:
    InvalidState;
```

This function supports the synchronization of the internal secure clock of the HSM with an external trusted clock. Such a synchronization usually works in two steps: (i) The HSM generates an unpredictable random challenge using its internal random number generator, starts an internal timer, and outputs the challenge. The challenge is sent to the trusted time source. (ii) The trusted time source generates a message that includes the current time t and the digital signature of the time source computed over the current time and the challenge of the HSM. This message is sent back to the HSM, which verifies the signature. If the signature is valid and the time τ elapsed since the generation of the challenge is shorter than a pre-specified threshold, then the HSM sets its internal clock to $t + \tau$.

The `getChallengeForTimeSynch` function is used in the above process to request the generation of the challenge. As the HSM has a single internal clock, it does not support parallel runs of the above protocol. Conceptually, the HSM can be in two different states: `normal` and `challenge_issued`. When the `getChallengeForTimeSynch` function is called in the `normal` state, the HSM generates a 128-bit challenge, stores it internally, starts its internal timer, outputs the challenge, and changes its state to `challenge_issued`. When the HSM receives a valid response from the trusted time source or its internal timer expires, it returns to the `normal` state and deletes the challenge from its internal memory.

An exception is thrown if the `getChallengeForTimeSynch` function is called when the HSM is in the `challenge_issued` state.

setTime

```
inputs:
    uint64 CurrentTime;
    byte* SignedHash;
    uint SignedHashLength;
outputs:
    --
exceptions:
    InvalidState;
    InvalidSignature;
```

This function is used in the time synchronization procedure to input the message of the trusted time source into the HSM. More specifically, a 64-bit timestamp value and the digital signature of the time source is input to the HSM. Recall that the digital signature must be generated over the timestamp value and the challenge received from the HSM. If the HSM is in the `challenge_issued` state, then it verifies the digital signature. If the signature is valid then the internal clock is set, the HSM returns to the `normal` state, and the challenge is removed from the memory.

An exception is thrown if the `setTime` function is called when the HSM is in the `normal` state, or the verification of the digital signature fails.

getTime

```
inputs:
    --
outputs:
    uint64 CurrentTime;
exceptions:
    --
```

This function can be used by the applications to read the value of the secure internal clock of the HSM.

getRandom

```
inputs:
    uint NumberOfBytes;
outputs:
    byte* RandomBytes;
    uint RandomBytesLength;
exceptions:
    --
```

This function is used to generate random numbers for cryptographic purposes, for instance, to be used as symmetric keys. This is needed when an application wants to encrypt a message with the supported ECIES hybrid encryption scheme where the message is first encrypted with a randomly generated bulk encryption key using a symmetric key cipher and then the random key is encrypted with the public key of the intended recipient. The random bulk encryption key should be generated by the HSM using the `getRandom` function. The encryption itself is then performed by the calling application, and once it is completed, the application must delete the random bulk encryption key.

initLongTermKeyUpdate

```
inputs:
    uint KeyID;
outputs:
    public_key* LongTermPublicKey;
exceptions:
    InvalidKeyID;
    NotEnoughMemory;
```

This function can be used to generate new long-term key pairs (long-term signature generation/verification or encryption/decryption key pairs). This is used when the certificate of the long-term key pair with keyID KeyID is nearly expired. The function only generates and temporarily stores the new private key and not removes the old key. The newly generated key cannot be used to sign or decrypt any message after this call. The function outputs the new public key, and sets the under_update flag of the key to true. If the function is called many times for the same keyID without calling `finalizeLongTermKeyUpdate`, only the last generated key is stored. The function outputs the newly generated public key.

An exception is thrown if there is not enough memory to store the new key, or the key is a short-term or root key, which cannot be updated with this function.

finalizeLongTermKeyUpdate

```
inputs:
    uint KeyID
    uint64 Timestamp;
    byte* SignedHash;
    uint SignedHashLength;
outputs:
    --
exceptions:
    InvalidSignature;
    InvalidTimestamp;
    NotUnderUpdate;
```

This function can be used to remove an old long-term key, and start to use the new key generated by the last call of `initLongTermKeyUpdate` with KeyID. This exchange of the keys is only comitted if it is authenticated by one of the stored long term root keys and is not too old. The input of the function is the keyID of the key to be updated, a timestamp and signed hash value of the string ”UPDATE LONG TERM KEYS”, the device identifier, the public key to be used, and the timestamp. After the successful execution

of the function, the new key can be used with the old keyID (the long-term signature generation key's ID is `0x0003`, and the long-term decryption key's ID is `0x0004`).

If the signature is not valid or the timestamp is too old, then an exception is thrown. If `initLongTermKeyUpdates` is not called yet, then an exception is thrown as well.

initDevice

```
inputs:
    byte[16] DeviceID;
    public_key[2] RootKeys;
    uint64 CurrentTime;
    byte* DeviceParameters;
    uint DeviceParametersLength;
outputs:
    public_key* LongTermPublicKeys;
    uint LongTermPublicKeysCount;
exceptions:
    AlreadyInitialized;
    UnknownAlgorithmID;
    InvalidPublicKeyValue;
    InvalidDeviceParameters;
```

This function is used to initialize the HSM. Initialization must happen only once. For this reason, after manufacturing, the HSM is in `not_initialized` state, and upon successful completion of the `initDevice` function, it changes its state to `initialized`, where calling the `initDevice` function again results in throwing an exception. Furthermore, initialization must happen in a secure environment where it is not expected that the communication between the HSM and the authority that performs the initialization can be attacked (modified or delayed).

The inputs of the `initDevice` function consist of a 128-bit globally unique device identifier, two long-term root public keys that are used to authenticate the commands issued by the authorities to the HSM, a 64-bit time value representing the current time, and a set of further device parameters including the threshold value used in the time synchronization procedure described earlier. Upon successful completion, the HSM installs the root public keys in its internal key data base and assigns the key identifiers `0x0001` and `0x0002` to them, sets its internal clock to the received time value, stores the device identifier and the device parameters, and changes its state to `initialized`.

Immediately after initialization, the HSM generates its long-term key pairs and outputs the public keys in an array of `public_key` data structures. These public keys can then be certified by the appropriate authorities. The HSM does not store certificates; this is the responsibility of the applications running on the OBU.

As we said before, an exception is thrown if the `initDevice` function is called in the `initialized` state. In addition, an exception is thrown if any of the supplied `public_key` data structures contains an unknown algorithm identifier or an erroneous public key value that cannot correspond to the given algorithm, or the supplied device parameters are invalid.

revokeRootKey

```
inputs:
    uint KeyID;
    uint64 Timestamp;
    byte* SignedHash;
    uint SignedHashLength;
outputs:
    --
exceptions:
    InvalidState;
    InvalidKeyID;
    InvalidTimestamp;
    InvalidSignature;
```

This function allows for the revocation of a long-term root public key. For this reason the caller must specify the key identifier (`0x0001` or `0x0002`) of the key to be revoked. In addition, the signature of the authority trusted for such a revocation together with a timestamp value must be supplied. The signature must be generated by the trusted authority with the private key that corresponds to the root public key to be revoked over the string ”REVOKE ROOT PUBLIC KEY”, the public key value, and the timestamp.

The HSM starts processing this call only if it has two valid root public keys, or in other words, if it is in the `two_root_keys` state. Otherwise, an exception is thrown. The HSM verifies the signature (with the root public key to be revoked) and the timestamp. If the signature is invalid or the timestamp is too old, then an exception is thrown.

Upon successful completion, the HSM deletes the specified root public key and changes its state to `one_root_key`. In this state, the HSM does not accept any more revocation commands.

setRootKey

```
inputs:
    public_key RootKey;
    uint64 Timestamp;
    byte* SingedHash;
    uint SignedHashLength;
```

```
outputs:
    --
exceptions:
    InvalidState;
    InvalidTimestamp;
    InvalidSignature;
    UnknownAlgorithmID;
    InvalidPublicKeyValue;
```

This function can be used to load a new root public key into the HSM given that it has a single valid root public key (i.e., it is in the `one_root_key` state). The caller must input the new root public key, a timestamp, and the signature of the authority that is computed with the private key that corresponds to the still valid root public key of the HSM over the string ”LOAD ROOT PUBLIC KEY”, the public key to be loaded, and the timestamp value.

The HSM starts processing this call only if it is in the `one_root_key` state, otherwise an exception is thrown. The HSM verifies the signature and the timestamp. If the signature is invalid or the timestamp is too old, then an exception is thrown.

Upon successful completion, the HSM installs the specified root public key and changes its state to `two_root_key`. In this state, the HSM does not accept any more load commands.

killDevice

```
inputs:
    uint64 Timestamp;
    byte* SingedHash;
    uint SignedHashLength;
outputs:
    --
exceptions:
    InvalidTimestamp;
    InvalidSignature;
```

This call is used to revoke the entire HSM. A need for such a revocation may arise if any of the long-term master private keys of the HSM is suspected to be compromised. The caller must input a timestamp and a digital signature computed by the authority with the private key that corresponds to one of the valid root public keys of the HSM over the string ”KILL DEVICE”, the device identifier of the HSM, and the timestamp. The HSM verifies the signature and the timestamp. If the signature is invalid or the timestamp is too old, then an exception is thrown. Otherwise, the HSM kills itself, meaning that it will not accept any more calls through its API.

Bundle description	Algorithm identifier
ECDSA with SHA-1 and elliptic curve domain parameters specified in [Cer00] under the name secp112r1	`0x0001`
ECDSA with SHA-1 and elliptic curve domain parameters specified in [Cer00] under the name secp112r2	`0x0002`
ECDSA with SHA-1 and elliptic curve domain parameters specified in [Cer00] under the name secp160k1	`0x0003`
ECDSA with SHA-1 and elliptic curve domain parameters specified in [Cer00] under the name secp160r1	`0x0004`
ECDSA with SHA-1 and elliptic curve domain parameters specified in [Cer00] under the name secp160r2	`0x0005`
ECDSA with SHA-1 and elliptic curve domain parameters specified in [Cer00] under the name secp256k1	`0x0006`
ECDSA with SHA-1 and elliptic curve domain parameters specified in [Cer00] under the name secp256r1	`0x0007`

Table 7.2: Description and identifier of the signature algorithm and parameter bundles supported in this version of the HSM API specification.

Bundle description	Algorithm identifier
ECIES with key derivation function ANSI-X9.63-KDF with SHA-1, MAC scheme HMAC-SHA1-80 (160-bit key), symmetric-key encryption scheme AES-CBC (128-bit key), standard elliptic curve DH primitive, and elliptic curve domain parameters specified in [Cer00] under the name secp112r1	0x0011
ECIES with key derivation function ANSI-X9.63-KDF with SHA-1, MAC scheme HMAC-SHA1-80 (160-bit key), symmetric-key encryption scheme AES-CBC (128-bit key), standard elliptic curve DH primitive, and elliptic curve domain parameters specified in [Cer00] under the name secp112r2	0x0012
ECIES with key derivation function ANSI-X9.63-KDF with SHA-1, MAC scheme HMAC-SHA1-160 (160-bit key), symmetric-key encryption scheme AES-CBC (128-bit key), standard elliptic curve DH primitive, and elliptic curve domain parameters specified in [Cer00] under the name secp160k1	0x0013
ECIES with key derivation function ANSI-X9.63-KDF with SHA-1, MAC scheme HMAC-SHA1-160 (160-bit key), symmetric-key encryption scheme AES-CBC (128-bit key), standard elliptic curve DH primitive, and elliptic curve domain parameters specified in [Cer00] under the name secp160r1	0x0014
ECIES with key derivation function ANSI-X9.63-KDF with SHA-1, MAC scheme HMAC-SHA1-160 (160-bit key), symmetric-key encryption scheme AES-CBC (128-bit key), standard elliptic curve DH primitive, and elliptic curve domain parameters specified in [Cer00] under the name secp160r2	0x0015
ECIES with key derivation function ANSI-X9.63-KDF with SHA-1, MAC scheme HMAC-SHA1-160 (160-bit key), symmetric-key encryption scheme AES-CBC (128-bit key), standard elliptic curve DH primitive, and elliptic curve domain parameters specified in [Cer00] under the name secp256k1	0x0016
ECIES with key derivation function ANSI-X9.63-KDF with SHA-1, MAC scheme HMAC-SHA1-160 (160-bit key), symmetric-key encryption scheme AES-CBC (128-bit key), standard elliptic curve DH primitive, and elliptic curve domain parameters specified in [Cer00] under the name secp256r1	0x0017

Table 7.3: Description and identifier of the encryption algorithm and parameter bundles supported in this version of the HSM API specification.

8 Bibliography

[Cer00] Certicom Research. SEC 2: Recommended elliptic curve domain parameters. Standards for Efficient Cryptography (SEC), version 1.0, September 2000.

[CPHL07] Giorgio Calandriello, Panos Papadimitratos, Jean-Pierre Hubaux, and Antonio Lioy. Efficient and robust pseudonymous authentication in vanet. In *VANET '07*, pages 19–28, New York, NY, USA, September 2007. ACM.

[DC85] David Chaum. Security Without Identification: Transaction Systems to Make Big Brother Obsolete. *Commun. ACM*, 28(10):1030–1044, 1985.

[FD05] Florian Dötzer. Privacy Issues in Vehicular Ad Hoc Networks. In *Workshop on Privacy Enhancing Technologies*, Cavtat, Croatia, May 2005.

[KSW06] Frank Kargl, Stefan Schlott, and Michael Weber. Identification in ad hoc networks. In *Hawaiian International Conference on System Sciences, HICSS 39*, Hawaii, USA, January 2006.

[MRJPH05] Maxim Raya and Jean-Pierre Hubaux. The security of vehicular ad hoc networks. In *SASN '05: Proceedings of the 3rd ACM workshop on Security of ad hoc and sensor networks*, pages 11–21, New York, NY, USA, 2005. ACM Press.

[PBH+07] P. Papadimitratos, L. Buttyan, J-P. Hubaux, F. Kargl, A. Kung, and M. Raya. Architecture for secure and private vehicular communications. In *ITST'07*, Sophia Antipolis, France, 2007.

[PMH08] P. Papadimitratos, G. Mezzour, and J.-P. Hubaux. Certificate revocation list distribution in vehicular communication systems. In *ACM VANET 2008*, San Francisco, CA, September 2008. (to appear).

[RCCKL06] C. Robinson, L. Caminit, D. Caveney, and Ken Laberteaux. Efficient Coordindation and Transmission of Data for Cooperative Vehicular Safety Applications. In *VANET 2006*, September 2006.

[RPJP06] M. Raya, P. Papadimitratos, and Hubaux J.-P. Securing vehicular communications. *IEEE Wireless Communications Magazine, Special Issue on Inter-Vehicular Communications*, October 2006.

[SKS+06] Elmar Schoch, Frank Kargl, Stefan Schlott, Tim Leinmüller, and Panos Papadimitratos. Impact of pseudonym changes on geographic routing in vanets. In *Third European Workshop on Security and Privacy in Ad hoc and Sensor Networks, ESAS 2006*, volume 4357 of *Lecture Notes in Computer Science*, Hamburg, Germany, September 2006.

[YCHKPL06] Yih-Chun Hu and Kenneth P. Laberteaux. Strong VANET Security on a Budget. In *escar 2006*, November 2006.