Project acronym:	EVITA
Project title:	E-safety vehicle intrusion protected applications
Project reference:	224275
Program:	Seventh Research Framework Program (2007–2013) of the European Community
Objective:	ICT-2007.6.2: ICT for cooperative systems
Contract type:	Collaborative project
Start date of project:	1 July 2008
Duration:	42 months

Deliverable D4.4.2: Test Results

Authors:	Yves Roudier, Hendrik Schweppe, Sabir Idrees (EURECOM); Ludovic Apvrille, Gabriel Pedroza (Institut Télécom);
Reviewers:	Dirk Scheuermann (Fraunhofer Institute SIT)

Dissemination level:	Public
Deliverable type:	Report
Submission date:	February 15th, 2012

Abstract

The objective of the EVITA project is to design, verify, and prototype an architecture for automotive on-board networks where security-relevant components are protected against tampering, and sensitive data are protected against compromise. Thus, EVITA will provide a basis for the secure deployment of electronic safety aids based on vehicle-to-vehicle and vehicle-to-infrastructure communication.

The most important tests, as specified in D4.4.1, based on a draft design of the security framework of the EVITA system were performed on the components actually integrated in the WP5000 prototype. These tests aim at controlling the security of the framework, both statically and dynamically. Some are more specifically safety-oriented in order to assess platform compromise risks while others are more security design-oriented and aim at detecting specific flaws in the design of security mechanisms that may have escaped previous analyses. This document describes the results of those tests.

Terms of use

Contents

List of tables

List of abbreviations

API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
C2X	Car to External Entities
CAN	Controller Area Network
CTP	Common Transport Protocol
ECU	Electronic Control Unit
FPGA	Field Programmable Gate Array
HSM	Hardware Security Module
LLD	Low Level Driver
MAC	Message Authentication Code
MCAL	Microcontroller Abstraction Layer
RPC	Remote Procedure Call
SPI	Serial Peripheral Interface
SWD	Security Watchdog Module

Document history

1.0	February 15th, 2012	Final version

1 Introduction

This deliverable gives a summary of the results of the tests conducted in the EVITA project and of their impact on the developments performed in other activities.

1.1 Test Objectives

The tests we performed in EVITA have two essential objectives: vulnerability testing and evaluating the correct implementation of the security design.

First and foremost, the first objective is to capture potential safety or security vulnerabilities that may put the developed security framework at risk of a compromise from an attacker, a rather challenging task even for specialists. We decided to focus on the exposed Application Programming Interface (API)s to account for the logical attack scenario put forward in EVITA, and in particular Car to External Entities (C2X) scenarios discussed in Task T2300. We also combined automated tests with manual code reviews to pin down faster potential weaknesses. The main difficulty of this objective in the EVITA architecture lies in the distributed nature of the embedded system and the number of components whose behaviors have been assessed together.

The second objective is to evaluate whether the implementation from Task T4000 shares the properties of the system defined in Task T3000. To this end, we essentially combined static code reviews with dynamic monitoring. The latter should intervene during runtime and we designed and implemented the log that supports this monitoring and traffic filters that assess the behavior of components. Code validation here means determining a normal profile that has to be fed to the filters, and evaluating the response of the filters themselves.

1.2 Results Interpretation

We have highlighted and corrected a few minor bugs in the framework implementation as well as discrepancies between the models proven in Task T3400 and the actual implementation. Those discrepancies essentially resulted from a change in the communication model, with the introduction of the EMVY Remote Procedure Call (RPC) which complicated the protocol stack design in the EVITA security framework. We have also put forward an integrated monitoring approach whose functionalities were validated on specific use cases.

The results of these tests should be taken with the usual word of caution about code validation: it is not meant to be exhaustive contrary to approaches conducted based on formal methods, which it actually complements with respect to areas where a formal proof would be overly complex to implement. Although we tried our best to identify interesting series of tests and validations with respect to our security objectives, it is not possible to evaluate all potential executions of the software in all deployment contexts.

1.3 Deliverable outline

We first describe tests performed on the Hardware Security Module (HSM) and on its integration with the Electronic Control Unit (ECU) in Section 2. We in particular discuss there issues arising from the need to automate tests performed on different interfaces in different execution environments. We then move on to examining security tests performed about communication protocols and their implementations in Section 3 through a combination of unit testing and manual code reviews. We finally discuss integration issues in Section 4, through both static and dynamic approaches, including at runtime.

2 HSM Testing

We discuss in this section our tests of the HSM functionalities from a security point of view. We undertook a combination of manual code reviews and fuzzing tests. The latter aims at establishing a more generic and automated testing approach, and is more original from a research perspective. Indeed, such tools are generally not used to validate the absence of flaws after designing a given architecture but rather as an attack tool. Those tests come in addition to basic unit tests, which we ran by hand from August 2011 on the HSM or Tricore platforms and whose major difficulty lies in the need to cross-compile rather than in the tests themselves. On the contrary, the distributed static security-related tests that we are focusing on in Task 4400 bring an interesting issue in that they involve testing complex and distributed embedded system that might trigger failures in different - and potentially deeply nested - software components.

The hardness of this assessment stems from the fact that one needs to evaluate the behavior of the complete chain consisting of the sender ECU, the EMVY stack on the recipient ECU, the protocol between the two EMVY stacks and application RPCs, the protocol between the Tricore board and the PowerPC on the Xilinx board in the recipient ECU, the firmware in the Xilinx board and finally the Field Programmable Gate Array (FPGA) on the Xilinx board. It is hard to do without such system-wide testing: our manual inspections of the code reveal that the Low Level Driver (LLD) software layer in the Tricore may constitute a threat for some data protected through the HSM/FPGA concept for instance. The use of the complete chain is here mandatory to be able to investigate all the idiosyncrasies of the implementation at hand. It is in particular extremely important to determine where vulnerabilities may lie in such a complex system and out of which component interactions they may arise. Since the complete chain is not implemented in the prototype (in particular while AUTOSAR and the LLDs where demonstrated separately, EMVY was not integrated into AUTOSAR), we focus on testing from the LLD drivers up to the FPGA implemented functions. Our tests consist in two parts: the first part is to be run on the HSM's PowerPC and performs random checks on the HSM API. The working combinations of parameters are then passed to the second part of the test in which those combinations are again tested on the Tricore to evaluate potential vulnerabilities in the LLDs or HSM communication stack.

The section is structured as follows: the rationale in subsection 2.1 places HSM Driver within the whole EVITA prototype architecture, justifies and briefly describes targeted tests. The scope and objectives of the work are precised in subsection 2.2. Right after, in subsection 2.3, an overview of the proposed testing environment is introduced. Subsection 2.4 summarizes the progress in tests and results. To ease description, the summary is split in general - for the overall Driver - and particular issues - for standalone Driver components. Finally, preliminary conclusions are presented in subsection 2.5.

2.1 Rationale

The Driver of the EVITA prototype specified in [6], is the tie between middleware applications, like EMVY and AUTOSAR, and lower HW layers, more specifically the Infineon Tricore board TC1797. Thus, the Driver provides an API to interact with the HSM [7]. From a security perspective, the Driver determines a first border between the

EVITA security anchor - the HSM -, and higher EVITA software components. Since that border is assumed within a trusted domain, the Driver grants many privileges to applications directly running on the top, e.g., full access to HSM buffers and cyphering keys handling. Consequently, the Driver is a security-sensitive component that should be preferably tested within its context, that is performing Integration and, in a second step, System Tests (see Deliverable [12]). As a mandatory stage, the basic functionality is first targeted, i.e., the mechanisms for passing data from top applications towards the HSM (named Driver Requests), and conversely (named HSM Callbacks). Later test stages go in depth by performing more elaborated testing, mostly based on Dynamic SW Analysis and Behavior Based Testing approaches, as described in [12]. Experimenting with Penetration Tests can also be an option, e.g., to evaluate the impact of malicious applications directly running on the top of the Driver. The scope and pursued objectives are better precised in next subsection.

2.2 Scope and Objectives

Amongst others, testing activities are intended to ensure that Driver behavior is predictable and secure: predictable means that responses from functions are as specified, i.e, according to the given stimuli. Secure means that behavior is free of weaknesses or issues that may lead to misuse or attack scenarios. Rather than being exhaustive, tests provide evidence of Driver operability, strengths and weaknesses, what helps to increase its reliability and also security. However, the assessment of Driver features is also part of WP4200 tasks, in which Driver modeling and formal verification were conducted [11]. Thus, Driver testing complements previous work by targetting dynamical features. Due to its nature, dynamical testing mostly depends upon SW operability, i.e., SW functionality. Hence, the exploration is conducted at three levels of abstraction as described below:

Coarse Level: Targets a standalone SW functionality or component - e.g., an API function - by using fixed parameters inside specification. Tests evaluation analyzes stimuli/response relationship and is limited to determine the final status of the SW component. More precisely, the EVITA codes returned by the function under test - e.g., `evitaResponseOk`, `evitaNotAvailable`, etc. - help to determine the final status of the function. The return codes obtained at LLD side are based upon ASN.1 specification. Consequently, the codes should correspond with the respective ones originally returned by the HSM. If no response at all is obtained after a given delay, the component is not operational. Even if some of these delays are not part of the EVITA specification, the response time of other functions was taken as reference. Coarse level tests are mainly performed during implementation of Driver.

Fine-grained Level: Targets one or more operable SW functions or components - e.g., execution of chained functions - by exploring the domain of parameter values accepted by the function(s). Evaluation of stimuli/response relationships is made by comparing expected and returned values. Indeed, once testing parameters are settled, an oracle is consulted to compute returned values, required in evaluations. In the EVITA prototype architecture, the HSM plays oracle's role. This level is suited to perform *Data Monitoring* as defined in [12].

Overall Level: Targets not only the operability of a set of SW functions or components, but their overall features with respect to a given scenario, e.g., wrong parameters injection. The domain of parameter values accepted by targetted SW components can be explored. Thus, along with stimuli/response relationships and an oracle, test case evaluations may require a set of criteria, defined along with the test scenario. Suitable testing categories at this level are: *Monitoring*, *Blackbox Fault Injection* and Penetration Testing [12].

According to previous descriptions, next objectives are settled:

1. Perform EVITA Driver testing in order to provide evidence about its operability, strongness and vulnerability, i.e., reliability and security
2. Rely on Coarse, Fine-grained and Overall tests to provide expected evidence
3. Settle and implement a Driver testing environment compliant with assessed testing levels
4. Design test cases within and outside the domain of accepted parameters
5. Achieve exploration of parameter domains
6. Design and automatically perform safety and security oriented tests cases
7. Achieve testing of the whole EVITA Driver API
8. Inform results and respective feedback

2.3 Driver Testing Environment

To achieve objectives stated in previous subsection (2.2), a testing environment is specified and implemented. More precisely, this environment contains two main applications (see figure 1):

HSM Fuzzer: This application directly interacts with the HSM API and thus is compiled for the PowerPC on the FPGA board, using the ELDK environment [2]. The HSM Fuzzer allows the execution of tests at three levels of abstraction, as they are defined in previous subsection. The test routines are intended to stress the HSM. Before performing a HSM call, the parameters of the respective API function are randomly chosen (fuzzing). Afterwards, the call is performed and the stimuli/response values are finally written in a file - referred as *C file*. More precisely, every line in the file makes the assignation of input/output values to array registers, using the C syntax. Stimuli/response instances constitute a base for comparisons which makes the HSM playing the role of oracle.

LLD Fuzzer: This application runs directly on the top of the Driver and thus is compiled with EB-Tresos [1] and Altium/VX-toolset [4] environments. The stimuli/response C file, generated at HSM side, is taken as a source. Indeed, for each stimuli/response instance, a LLD request is created using the same stimuli parameters. The request

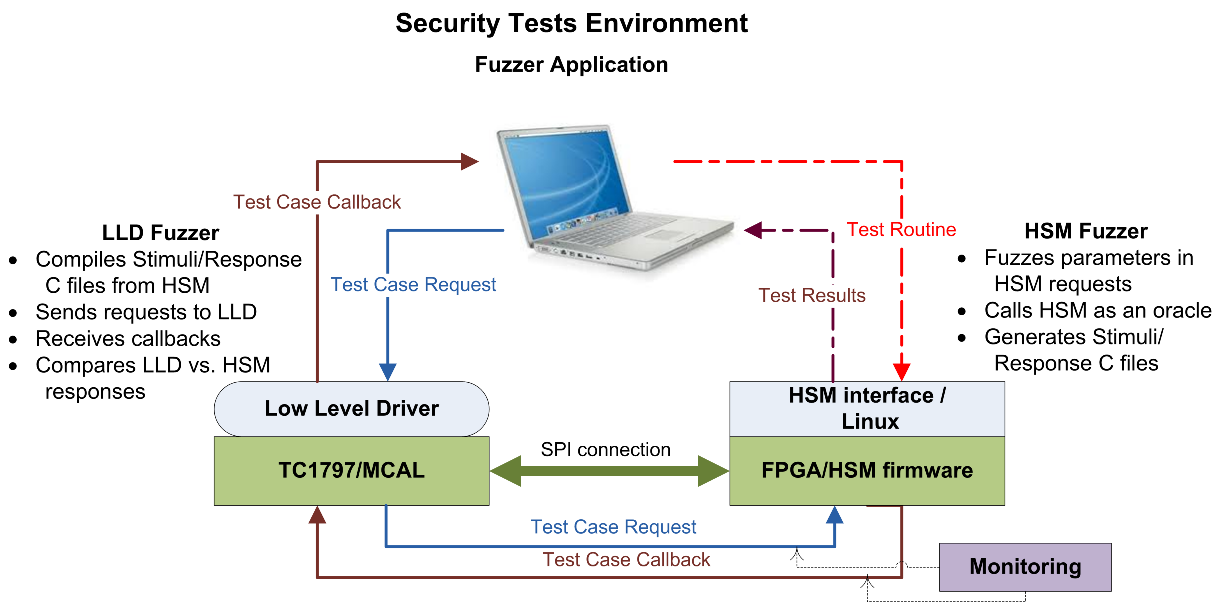


Figure 1 Driver Tests Approach

is afterwards sent to the LLD. Once obtained, the response is compared with the respective values from the reference C file. The evaluation of HSM vs. LLD instances is automatically conducted by setting breakpoints at corresponding comparisons whenever LLD and HSM values do not match. Finally, exchanges between Tricore and FPGA boards via Serial Peripheral Interface (SPI), are monitored and stored in a log file for further test case analysis.

The nominal execution of a testing routine is as follows: The defined test case may target one or more HSM API functions. Each parameter within a function call is fuzzed by generating a random value from a seed. To cover both inside and outside specification testing, random values can be mapped to predefined intervals, using for instance the modulus function. Once set, call parameters are written into a C file and the request is sent to the HSM. Eventually, returned values are also written in the C file, thus defining a stimuli/response instance. Once the test case is finished on HSM side, one or more C files are generated and integrated as part of the LLD Fuzzer. Indeed, after compilation and flashing on the Tricore - using the HiTOP debugger [3] -, the LLD Fuzzer executes a set of LLD requests thus reproducing the test case. A comparison between HSM and LLD responses is right after performed. Relying on the HiTOP debugger, breakpoints are set at unsatisfied comparisons what automatically points out differences between HSM and LLD responses. Test case analysis is complemented by monitoring HSM behavior during LLD Fuzzer execution.

As shown in figure 2, the testing implementation relies on a blackbox approach: the targetted components receive certain stimuli and return a response. To conduct evaluation of test results, stimuli/response instances are characterized with respect to next parameters:

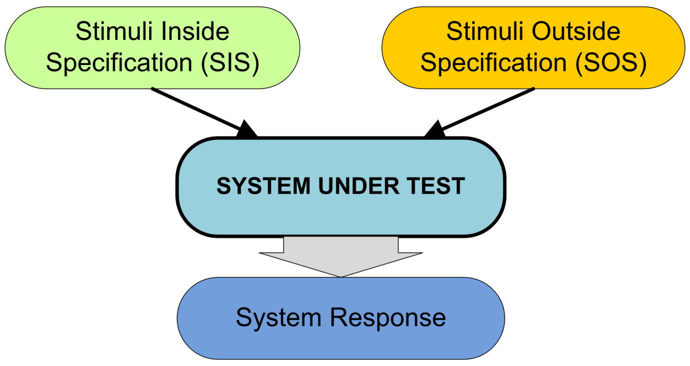

Stimuli Inside Specification (SIS)
Stimuli Outside Specification (SOS)
SYSTEM UNDER TEST
System Response

Figure 2 System under blackbox testing

SIS: Stimuli Inside of Specification

SOS: Stimuli Outside of Specification

SR: System Response

Next definitions are adopted for evaluation of system response with respect to provided stimuli:

System Correctness: Both the stimuli provided to the target system and the corresponding response are as specified.

System Inconsistency: The target system receives a stimuli inside specification (SIS) but the response is not as specified.

System Robustness: When a stimuli outside of specification (SOS) is received, the system continues its nominal operation on other requests.

Finally, next assumptions have been taken into account for a correct evaluation of results in our testing framework:

A1: The Evita HSM API and related architecture can play the role of the oracle.

A2: Tool chain for compilation of HSM implementations is bug free.

A3: Microcontroller Abstraction Layer (MCAL) Drivers on Tricore, SPI communication (Tricore$< - >$HSM) and other EVITA architecture components are bug free and are configured according to specification [7].

A4: Tool chain for compilation and flashing of LLD implementations is bug free, i.e., Tresos [1], VX-toolset [4], HiTOP [3].

2.4 Tests and Results

Next subsections describe the results that have been obtained using the previously defined testing environment.

2.4.1 Test Cases Description

Initially and during Driver implementation, the Driver API was tested at a coarse level, i.e., to determine its operability. Along with that, further overall tests have been semi-automatically conducted. Thus, performed tests are as follows:

Sequential Requests: A sequence of requests is sent to the same Driver function using different Abstract Syntax Notation One (ASN.1) parameters. There is no delay between each request. Also, both received calls and returned responses are monitored at HSM side. Sequential calls prove that access to shared buffers is mutually exclusive, e.g., in the Request Queue, Request List, and Communication buffers (HOST Buffer).

Chained Requests: Sequences of chained requests such as `MacInit()`, `MacUpdate()` and `MacFinish()` are targetted. As specified in [16] and [7], a session is opened by the HSM during processing of chained requests. The behavior of mechanisms managing linked requests is explored, e.g., non-closed HSM sessions. Both received calls and returned responses are monitored at HSM side.

Faulty Parameter Requests: Driver API functions are called using parameters out of HSM specification, e.g., wrong HSM sessions, nonexisting or wrong key handles, etc. Parameters outside of Driver specification are also used, e.g., wrong EVITA enumerations or structures. The whole Driver API is thus targetted. Both received calls and returned responses are monitored at HSM side.

HSM and LLD Fuzzing: The engines for fuzzing the chained API functions `CipherInit()`, `CipherProcess()` and `CipherUpdate()`, have been implemented on HSM and LLD sides. The engines for fuzzing the whole API functions are still in progress. Fuzzing is a technique that can combine all test patterns described above.

2.4.2 Results: General Issues

According to tests results, next general issues have been identified in the overall behavior. Testing is based upon final versions of EVITA Driver code - released without version number - and HSM firmware - version v0.6.5.

1. Sequential requests may be overwritten in Driver buffers, i.e, the Driver is, up to now, unable to grant mutually exclusive access to shared buffers. This LLD weakness is identified by performing sequential tests and monitoring at HSM side, and directly proved with the fuzzing tester.

2. Previous issue lead us to statically analyze the code. From that analysis, we could conclude that the Request Queue is the only buffer protected with access locks. However, the mechanisms for mutual exclusive access are not yet defined.

3. The static analysis of code also demonstrated that a mutual exclusive access is also required in the following shared buffers:

   (a) Request List

   (b) Serializing/deserializing local buffers (HOST Buffer)
   (c) Globally defined variables, e.g., buffer level counters

4. As a consequence of previous points, the Driver does not properly implement the multiple user session approach: user sessions are managed via a transaction ID that is sent within LLD calls and callbacks. However, the leakage in protection of shared buffers, may lead to wrong user session management, e.g., by overwriting an entry in the Request List.

5. The sequence of session handles, generated by the HSM to control chained requests, is repeated when the HSM is re-started, what increases the chance for guessing a valid session handle. This was observed both during sequential and chained requests tests, since the HSM must be manually re-started when the maximum number of sessions is reached.

6. If the maximum number of HSM sessions is reached, no more requests are accepted by the HSM, even if some HSM sessions remain open. This issue was initially identified in sequential and chained requests tests and afterwards proved in the fuzzing approach.

7. If a Driver request for finishing a chained sequence is corrupted or overwritten, the respective HSM session remains open forever. This may be observed by running fuzzing routines with a huge number of calls.

8. The Driver mainly targets functional issues related with the prototype implementation and in regard to show cases.

9. All applications directly running on top of the Driver are allowed to use the whole LLD API, and privileged access to HSM API.

10. Every application running on top of the Driver can use HSM session handles: there is no association between requesting applications and HSM session handles. This is concluded after an analysis of stimuli/response instances in chained requests tests.

11. Relying on the overall test results and analyses, we came to the conclusion that there is no mechanism in the Driver to react in case of:
    (a) Full Request Queue
    (b) Full Request List
    (c) Long delays of HSM responses
    (d) Unanswered Driver requests (LLD livelocks)
    (e) Unclosed HSM sessions (HSM livelocks)
    (f) No available sessions on the HSM (Denial of Service)
    (g) Re-initialization of the HSM

2.4.3 Results: Particular Issues

Table 1 presents results of tests conducted at a coarse level (see subsection 2.2). Each row includes a reference to a Driver API function, respective result and comments. More precisely, return codes within second column are obtained using parameters inside of specification. Thus for instance, the `evitaReturnOk` code indicates that the call was accepted and successfully processed by the HSM. Every Driver API function was individually tested with all algorithm identifiers defined in EVITA-ASN.1. However, by the time this report is written, some of those algorithms are not fully implemented on the HSM. Many values outside of specification were also used. The use of parameters outside of specification stresses the API function and the whole LLD/HSM implementations. Thus, it is proved whether the LLD and HSM can properly deal with wrong input values, i.e., by correctly identifying and signaling the respective error(s) relying on specified EVITA-ASN.1 return codes (error handling). Along with that, the LLD and HSM should continue their normal operations on other requests, what proves LLD/HSM robustness for the test routine. Since the HSM provides a maximum number of sessions for chained functions, this maximum is some times reached during test routines, mainly when an initial chained function is called many times, e.g., with 5000 calls. In such case, only invalid session handles can be used afterwards. Of course, tests with randomly generated session handles prove LLD behavior beyond the maximum allowed by the HSM. Moreover, random session handles test the multiuser approach by adding calls that might match with an already opened HSM session, thus impersonating the original caller. Finally, all the tests were conducted based upon final versions of EVITA Driver code - released without version - and HSM firmware - version v0.6.5.

Table 1 Results of coarse level Driver tests

Driver Request	**Return Code**	**Comments**
CipherInit()	evitaReturnOk	The request was tested with all algorithm identifiers defined in ASN.1 specification. 10 HSM sessions are available
CipherProcess()	evitaReturnOk	The function was tested with randomly generated application identifiers and invalid HSM sessions. The values of max_chunk_size and chunk_block_size set by CipherInit() are not mandatory and can be modified
CipherFinish()	evitaReturnOk	The function was tested with randomly generated application identifiers and invalid HSM session handles. The HSM session can be closed before message ciphering is completed
MacInit()	evitaReturnOk	Several parameters were tested according to ASN.1 specification

Continued on next page

Table 1 Results of coarse level Driver tests – continued from previous page

Driver Request	Return Code	Comments
MacUpdate()	evitaReturnOk	The function was tested with randomly generated application IDs and invalid HSM session handles. The values of max_chunk_size and chunk_block_size set by MacInit() are not mandatory and can be changed
MacFinish()	evitaReturnOk	The function was tested with randomly generated application IDs and invalid HSM session handles. The HSM session can be closed before message mac computation is completed
HashInit()	evitaReturnOk	Several parameters were tested according to ASN.1 specification
HashUpdate()	evitaReturnOk	The function was tested with invalid session handles. The HSM session handle can be known and used by all applications running on top of the Driver. The values of max_chunk_size and chunk_block_size set by HashInit() are not mandatory and can be changed
HashFinish()	evitaReturnOk	The function was tested with invalid session handles. The HSM session can be finished even before message hashing is completed
HashFinishAndExtend()	Not tested yet	
SignInit()	evitaReturnOk	All algorithm identifiers were tested
SignUpdate()	evitaReturnOk	The function was tested with invalid session handles. The HSM session handle can be known and used by all applications running on top of the Driver. The values of max_chunk_size and chunk_block_size set by SignInit() are not mandatory and can be changed
SignFinish()	evitaReturnOk	The function was tested with invalid session handles. The HSM session can be finished before message signature is completed
VerifyInit()	evitaReturnOk	Several parameters were used for testing according to ASN.1 specification
VerifyUpdate()	evitaReturnOk	The function was tested with invalid session handles. The HSM session handle can be known and used by all applications running on top of the Driver

Continued on next page

Table 1 Results of coarse level Driver tests – continued from previous page

Driver Request	**Return Code**	**Comments**
VerifyFinish()	evitaReturnOk	The function was tested with invalid session handles. The HSM session can be closed before message verification is finished
RngGetRandom()	evitaReturnOk	The function was tested for several data sizes. Only 1 pseudo random algorithm is implemented
CreateCounter()	evitaReturnOk	A maximum number of counters is established (currently only 2)
ReadCounter()	evitaReturnOk	The function was tested with sequential calls and invalid counter IDs.
IncrementCounter()	evitaReturnOk	highWord and lowWord attributes in the counter are never modified
DeleteCounter()	evitaReturnOk	The function was tested with invalid counter IDs
ModuleStatus()	No response at all	Several parameters were used according to ASN.1 specification. The HSM never receives the request
SelfTest()	evitaNotAvailable	The function was tested using all algorithm identifiers according to ASN.1 specification
CreateRandomKey()	evitaReturnOk	Several parameters were used according to ASN.1 specification
CreateDhKey()	evitaNotAvailable	Several parameters were used according to ASN.1 specification
KeyExport()	evitaReturnOk	Tests targeting several exportable keys were performed
KeyImport()	evitaReturnOk	The *KeyImport()* requests can be replayed. The same key can be imported several times but with different session handles
KeyRemove()	No response at all	Symmetric and asymmetric keys were targeted. The HSM never receives the request
KeyStatus()	evitaReturnOk	Symmetric and asymmetric keys were tested
ExtendEcr()	evitaReturnOk	Several ECU configuration registers were extended
RetrieveEcr()	asnTypeConversion-Error	Several ECU configuration register indexes were targeted
PresetEcr()	escWhirlpoolUpdate Error	Several ECU configuration register values were used
CompareEcr()	evitaReturnOk	Several parameters were used
CreateTimeStamp()	evitaClockNot Synchronized	Several parameters were used. Synchronization is required

Continued on next page

Table 1 Results of coarse level Driver tests – continued from previous page

Driver Request	**Return Code**	**Comments**
CheckTimeStamp()	evitaClockNot Synchronized	Several parameters were used. Synchronization is required
GetTimeSyncChallenge()	evitaReturnOk	
SetUtcTime()	evitaUtc SynchronizationFailed	Several parameters were used. A procedure for synchronization should be executed
GetUtcTime()	evitaUtc SynchronizationFailed	Several parameters were used. A procedure for synchronization should be executed
GetTickCount()	evitaReturnOk	

2.5 Conclusions

An approach for testing EVITA prototype Driver has been presented. The approach aims to provide evidence of Driver operability, strongness, weaknesses and thus, about its reliability and security. Dynamical tests target SW components and are defined at three levels of abstraction: Coarse, Fine-grained and Overall. Tests at Coarse and Fine-grained levels mainly target behavior of standalone Driver components, whilst tests at Overall level target more abstract features, e.g., Driver security. Relying upon declared levels, a Testing Environment was envisaged. The environment is intended to automatically perform tests and evaluate results by interacting with the EVITA prototype architecture, in which the HSM plays a role of oracle. The engines for fuzzing three chained API functions have been coded and successfully implemented on HSM and LLD sides. Even if the testing environment is still work in progress, several tests have been already conducted, mainly at Coarse and Overall levels. According to results and despite identified issues, the EVITA Driver is ready for showing purposes, i.e., it is suitable for prototype demonstrations. However, in our opinion, **it is mandatory to consider identified issues for improving operability, reliability and security of EVITA Driver**. Further tests should be conducted in order to go in depth with exploration of parameter domains and other interesting test cases. Thus, the Testing Environment should be finished to cover the whole HSM and LLD APIs and to automatically execute and evaluate more complex routines and cases.

3 Protocols Assessment

We performed code validation on two different types of protocols: purely on-board protocols and car-to-infrastructure protocols that are required in order to configure the car and more specifically its security policy.

On-board protocols are all defined on top of the RPC like interface defined in EMVY, contrary to the original design we described in Deliverable D3.3 [14] This means that the security of the design and implementation of the RPC, which was not studied in formal method based reviews is quite crucial here. We essentially undertook code reviews on the RPC and its integration with various components that unearthed a design problem regarding cross-layering that was subsequently corrected in the framework implementation as well as various bugs.

Regarding the second type of protocol, we performed more usual tests of the protocol including manual fuzzing of the inputs. This already highlighted some weaknesses in the integration of the ASN.1 parser in the EVITA framework in particular, which should be addressed in order to prevent potential vulnerabilities (which we did not find) or plain denial of service attacks.

3.1 RPC Security Integration

Th EMVY RPC library allows applications to use functionality on the client itself and also to access higher-level security functionality through the master node. This is achieved using an RPC layer by encapsulating function requests in specially crafted ASN.1 encoded request and response packets. The EMVY RPC Layer is based on several underlying security components (i.e., CCM, EAM, PDM, KMM, etc.). These security components are necessary for a client to securely communicate itself to a service, and vice versa, in each call and reply message. The specified RPC model allows EMVY clients to invoke several security services (i.e., login, logoff, security event notice, etc.) from client to server as part of the RPC invocation. The EMVY server can then discover the client's identity and authorization credentials, and determine what access to authorize. However, some permissions relate to operations offered by clients through the RPC mechanisms (including EVITA communication and security mechanisms). The need to authorize operations based on RPCs from EMVY/EVITA together with the fact that only channels, not RPC messages are authenticated has forced us to piggyback the transport-level authentication on internal framework calls from components like the CCM up to the application layer.

3.1.1 RPC Security Challenges

This section describes an example of an actual exploitation of the weaknesses in EMVY RPC design as described above. Conceptually variant of attacks are possible including denial of service (particularly logoff other entities) and impostering valid users. This example describes a login/logoff attack. The attack is accomplished by using the `EMVY_logoff_entity()` service provided by EMVY RPC. This is used to logoff the already authenticated entity so that the intruding entity could stop all the services provided/accessed by an entity. Disabling any ECU while services are running may cause safety critical problems, depending on the function ECU is responsible for. In our RPC level assessment setup we have created two entities. The *client* 1 is used to login to the machine. The second entity is given a *client* 2 as an identifier which acts as an intruder entity. The transcript in Figure 3 shows the use of the two RPC request from *client* 1 to infiltrate EMVY RPC level security in order to `logoff()` *client* 2.

```
Entity (const EMVY::String & description,
        const EMVY::String & identifier,
        const EMVY::String & issuerIdentifier );
```

Listing 1 EMVY Entity data structure

In the first step, RPC request is sent from *client* 2 for authentication and invocation of RPC `EMVY_login_entity()` service. After successful authentication, the intruder may generates a `EMVY_logoff_entity()` request with a fake entity data structure (as shown in List 1) that looks like *client* 1 and sends it to the EMVY master node. This requires that the intruding entity (*client* 1) be in the network, may have knowledge about other entities, or have capabilities for scanning all connected entities in the network. Since authentication is only performed at the transport layer and not further considered at the RPC level, *client* 1 or any other intruder entity on a network could easily create a fake RPC message simply by pretending to be *client* 2. Whereas, master node only verifies that *client* 1 is in its "EntityAuthenticationList". If its so, it removes the *client* 1 from EntityAuthenticationList and close the connection with client 1.

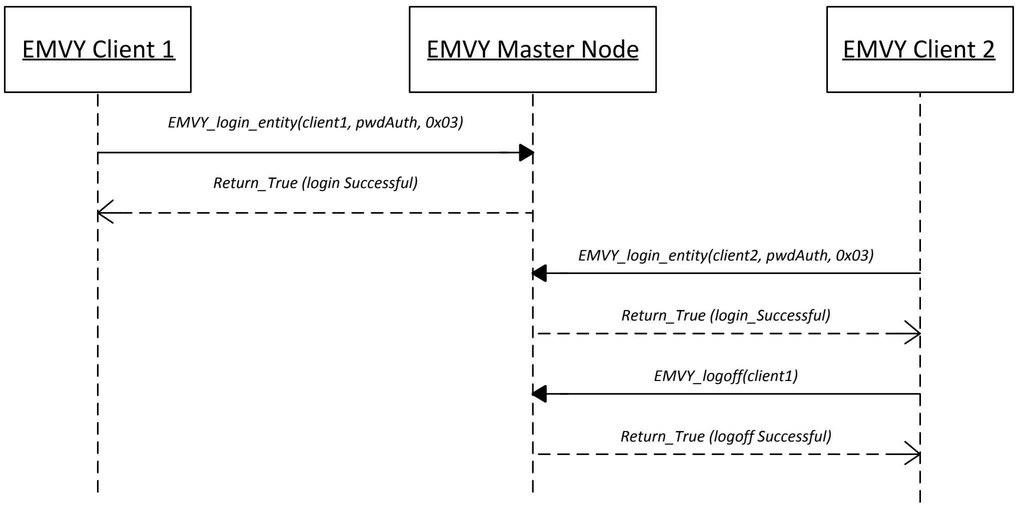

EMVY Client 1
EMVY Master Node
EMVY Client 2
EMVY_login_entity(client1, pwdAuth, 0x03)
Return_True (login Successful)
EMVY_login_entity(client2, pwdAuth, 0x03)
Return_True (login_Successful)
EMVY_logoff(client1)
Return_True (logoff Successful)

Figure 3 RPC Login-Logoff Attack Scenario

The security problems that result are due to the semantic meaning of the RPC services. For instance, the RPC service `EMVY_logoff_entity(const Entity* entity)` only require `Entity` as a parameter.

3.1.2 RPC Security Fixes

An obvious solution to this problem would be to change the signature of `logoff` RPC service, by adding authentication ticket parameter (`EMVY_logoff_entity(const Entity*`

`entity, AuthenticationTicket* authentication_ticket`[1])) in the method. However this brings about an implementation design restrictions. Under current systems we believe this can only be accomplished by binding transport layer authentication and using these authentication tickets at the RPC layer. This would allow EMVY layers to bind RPC services invocation with transport layer authentication. An authentication ticket obtained through transport layer should be used when `logoff()` RPC service is called. We used an attribute-value-assertion (AVA) list as a data structure to convey this information (see List 2). This proved to be a flexible and modular way to bind with the existing EMVY layers.

```
SecurityObject *ava = new AVAList(client->getRemoteEntity(),object->
    getObjectDescription(), object->getObjectIdentifier(), ctx);
```

Listing 2 Attribute-Value-Assertion (AVA) Integration

Another modification introduced by enforcing RPC level access control that defines and restricts the behavior of the RPC in EMVY/EVITA master and clients. Such access control aims in particular at controlling stateful operations like a login, a logoff, setting the security policy, etc. As mentioned above, operations are granted based on the access rights defined for each EMVY client and based on the contextual/environmental information stored in the AVA list (see List 3). This could be applied in varying degrees.

```
switch ( rpc_req->getPayloadCommandPR() ) {
case Asn1EmvyRpcInterface__payload__emvylibCommand_PR_emvyLoginEntity
    : {
SharedPtr<Entity> entity = rpc_req->getEntity();
enum EAMAuthenticationPluginTypes method = rpc_req->getMethod();
SharedPtr<AuthenticationTicket> authenticationTicket = rpc_req->
    getAuthenticationTicket();
LoginContext ctx = rpc_req->getLoginContext();
SecurityObject *object = new SecurityObject("logIn", 0x10000000 );
 SecurityObject *ava = new AVAList(client->getRemoteEntity(),object->
     getObjectDescription(), object->getObjectIdentifier(), ctx);
SecurityOperationSet *operation = new SecurityOperationSet(1) ;
SharedPtr< ReturnCode > ret = emvylib.EMVY_request_authorization (
    entity, ava, *operation, NULL);
   if(ret->getCode() == 0)
    {
     EMVY_DEBUG<<"Login Operation Authorized"<<std::endl;
    } else  {
       EMVY_DEBUG<<"Operation Not Authorized"<<std::endl;
    }
break
```

Listing 3 Attribute-Value-Assertion (AVA) Integration

3.2 Security Policy Distribution Protocol

Security policies constitute an important part in the EVITA security framework in that they constitute a description of an interface with the vehicular network behavior for

[1]Generated once the entity is successfully `login()`.

security administrators at the different stakeholders. In particular, a security policy will be defined and will evolve with the on-board network features. Assuring the correctness of security policies is becoming an important and challenging task. Identifying discrepancies between security policies and correct enforcement by EVITA security framework is based on the premises that the policy specification and encoding is done correctly. To evaluate the correctness of the security policy specification and policy distribution protocol, we have applied several existing test approaches.

Errors in policy specifications may be discovered by leveraging existing techniques for software testing such as mutation testing, which involves modifying security policies in small ways. Mutation testing which is a specific form of fault injection that consists in creating faulty versions of policy by making small semantic or syntactic changes. These so-called mutations, are based on well-defined mutation operators that either mimic typical encoding errors such as, specifying the wrong length, using the wrong ASN1_TYPE operator or involving incorrect use of the XACML logical constructs. Based on the mutation testing approach, we analyzed and verified whether a security policy, during serialization (at the backend system) and deserialization process (in the vehicle), is resistant against such fault injections.

- Using ASN.1 compiler for (de)serializing policies, one can exploit the bug to cause an out-of-bounds read operation, most likely resulting in a denial of service. A malformed or unusual ASN.1 tag value can trigger this issue.
- Invalid ASN.1 encodings that are rejected by the parser may potentially trigger a memory-management error.
- Incorrect logical construct include policy or rule combining algorithms, policy evaluation order, rule evaluation order and various functions found in conditions.

3.3 Conclusion

We have presented the results of the security test for security policy distribution protocol and RPC level security assessment. We showed how different parts of policy distribution protocol can be vulnerable to attacks. These vulnerabilities are mostly due to the hard assumptions made on the underlying asn.1 compilers. Thus, we assume that proper exception handling mechanisms should be considered during implementation of asn.1 based security policy protocols. Furthermore, we have identified several design flaws during protocol assessment. We essentially undertook code reviews on the RPC and its integration with various components that unearthed a design problem regarding cross-layering that was subsequently corrected in the framework implementation as well as various bugs. We have presented several solution for fixing these vulnerabilities and implementation bugs such as, we showed how low level authentication can sensibly be linked with specific authorization at upper layers, in order to protect system from performing unauthorized operations.

4 System Level Validations

4.1 Performance Measurements

Despite the absence of a full-fledged deployment in a car, we used simulations in order to gain some understanding about the stress imposed by our communication security mechanisms on the bus systems. In order to provide real world results for message transfer times of messages that include Message Authentication Code (MAC)s on standard automotive CAN buses, we modeled three nodes on a CAN bus in TrueTIME 2.0 [5], a Matlab-Simulink toolkit, which supports the simulation of CAN networks. One node generates high-priority background noise at 60%, which will always be prioritized over our payload (which might be considered a pessimistic assumption).

The framework allows to attach custom nodes onto a simulated network (the CAN bus in our case). We used the `networked` simulation setup in Simulink (depicted in Fig. 4). We have implemented the transport protocol including a MAC truncation mechanism and measured the real-world latency computed by the simulation environment.

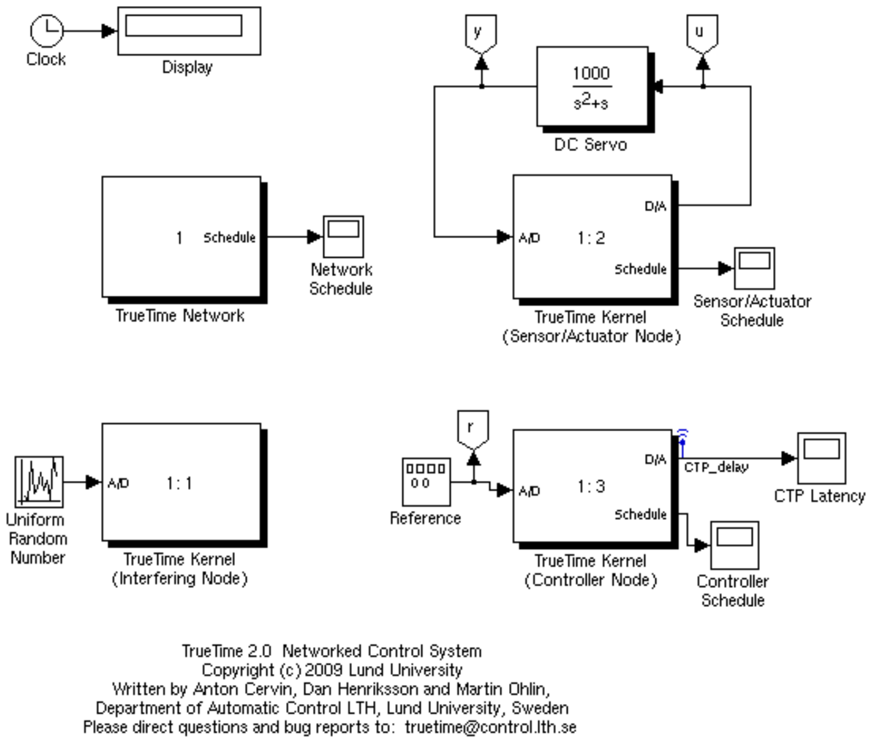

Clock
Display
y
u
1000
s2+s
DC Servo
1 Schedule
Network Schedule
TrueTime Network
A/D
D/A
1:2
Schedule
TrueTime Kernel (Sensor/Actuator Node)
Sensor/Actuator Schedule
Uniform Random Number
A/D 1:1
TrueTime Kernel (Interfering Node)
r
0000 00
Reference
A/D
D/A
1:3
Schedule
CTP_delay
CTP Latency
TrueTime Kernel (Controller Node)
Controller Schedule
TrueTime 2.0 Networked Control System
Copyright (c) 2009 Lund University
Written by Anton Cervin, Dan Henriksson and Martin Ohlin,
Department of Automatic Control LTH, Lund University, Sweden
Please direct questions and bug reports to: truetime@control.lth.se

Figure 4 Simulink setup with the TrueTime toolbox to simulate CAN bus. The interference node is used to generate bus load. The transport protocol simulation including message segmentation is implemented between the controller node and the sensor/actuator node.

We have conducted a number of tests at different bus payloads, in order to show the protocol feasibility even on halfway saturated buses. The results can be seen in Figure 5.

Our measurements for end-to-end message latency at 60% payload can be found in

Table 2. As the behavior of a busy CAN bus is rather non-deterministic, we included the minimum, maximum, and average delays, that we measured over 100 probes for each MAC length given. It can be seen that our security header does not significantly impact the end-to-end latency.

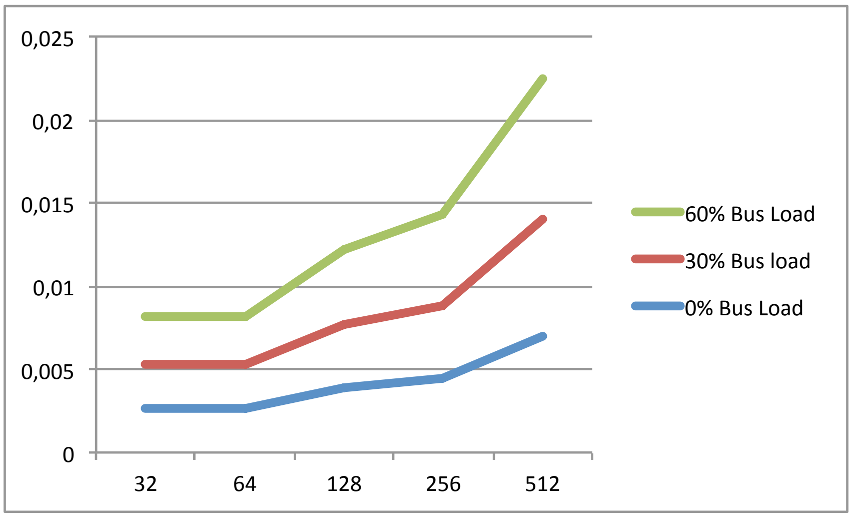


Figure 5 End-to-end latency for truncated MACs

4.2 Bootstrap

As we previously mentioned, the bootstrap is a pivotal function for enabling security in the vehicle and would need to be tested in order to ascertain that the overall on-board network does not incorporate untrusted components and that the platform has not been tampered with. It consists in two components. The secure boot sequence occurs at every ECU and involves interactions between the local CPU and the local flash memory; it would be the target of attackers having physical access to an ECU. In contrast, the trusted boot process relies on communication between the ECUs and would be the target of attackers able to perform a remote compromise on an ECU, this latter scenario being the one assumed in EVITA. We finally were not able to test this process as the secure boot demonstration and EMVY based communication demonstration were implemented separately. Yet we reiterate our recommendation that **this particular bootstrap function would mandatorily need to be tested in any product deployed based on a similar architecture**.

MAC/bits	0	32	64	128	256	512
Minimum	0.0004	0.0007	0.0011	0.0016	0.0027	0.0059
Average	0.0011	0.0018	0.0021	0.0030	0.0041	0.0073
Maximum	0.0030	0.0041	0.0043	0.0050	0.0060	0.0090

Table 2 Latency of CTP CAN packets with eight bytes payload in seconds. Bus is at 60% load with higher priority traffic.

4.3 Security Policy

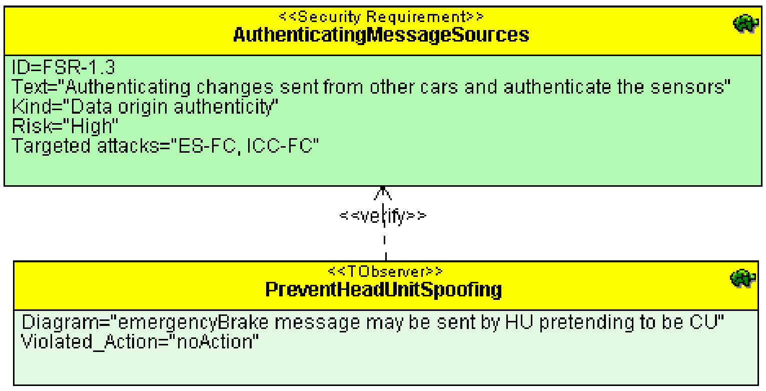
We applied the structural coverage criteria approach proposed in [8] to identify conflicts in security policies. Following this approach, we observed the generation of authorization tickets, their configuration, and the evaluation of different elements of such authorization tickets during the processing of authorization request and response. For instance, if any request is not evaluated against an authorization ticket during testing, then potential errors in that ticket cannot be identified. The tool distributed by the authors generates XACML requests and cannot therefore be used for testing our ASN.1 encoded requests. Given the small number of policy rules defined in the demonstrator, we instead followed this method manually to produce our tests.

- **Authorization Ticket coverage:** An authorization ticket is covered by a request if the authorization ticket is applicable to the request and the authorization ticket contributes to the decision. Authorization ticket coverage is the number of covered ticket divided by the number of total tickets loaded into the PDM. We further evaluate the authorization ticket by evaluating different elements defined in the authorization ticket.
  - *Subject Coverage*: A subject for an authorization ticket is covered by an entity (EMVY entity) authorization request if the subject is also applicable to the entity request and the authorization ticket contributes to the decision; in other words, the authorization ticket is applicable to the request and all the conditions (i.e, subject attribute values) in the subject are satisfied by the request and the PDM has yet to fully resolve the decision for the given request. Subject coverage is the number of covered EMVY entities divided by the number of total subjects.
  - *Resource Coverage*: A Resource for an authorization ticket is covered by an object (EMVY object) authorization request if the resource is also applicable to the object request and the authorization ticket contributes to the decision. Resource coverage is the number of covered EMVY Objects divided by the number of total resources.
  - *Action coverage*: An action for an authorization ticket is covered by an operation (EMVY SecurityOperation) authorization request if the resource is also applicable to the operation request and the authorization ticket contributes to the decision. Action coverage is the number of covered EMVY Security Operations divided by the number of total actions.
  - *Rule Coverage*: The evaluation of the condition for a rule has two outcomes: true or false. A true condition for a rule is covered by a request if the rule is covered by the request and the condition is evaluated to be true. A false condition for a rule is covered by a request if the rule is covered by the request and the condition is evaluated to be false. Condition coverage is the number of covered true conditions and covered false conditions divided by twice of the number of total conditions.

The final architecture of the demonstration was substantially changed with respect to our original expectations and finally did not include any gateway incorporating the PDM (and security policy engine) as a filtering device (whereas policies might describe such a filtering). We instead performed unit testing on the policy engine which led to only a few bug fixes.

4.4 Requirements Validation on Code

Security requirement validation is recognized as a necessary condition for security assurance of the system. Requirement validation is part of requirement traceability property. Thus, we define requirement traceability if (i) the origin of each of its requirements is apparent and if (ii) it facilitates the referencing of each security requirement in different phases of software development life cycle. In our proposed methodology [9, 13], security requirements are described in a way that relates to use cases, attacks and to models of the system. Therefore, we provide a way to trace security requirements w.r.t. other system elements. Security requirements also contain observers (as shown in figure 6), which may be seen as test cases meant to be used for the formal verification, or simulation, or during system testing (code validation) phase.



ID=FSR-1.3
Text="Authenticating changes sent from other cars and authenticate the sensors"
Kind="Data origin authenticity"
Risk="High"
Targeted attacks="ES-FC, ICC-FC"

Diagram="emergencyBrake message may be sent by HU pretending to be CU"
Violated_Action="noAction"

Figure 6 SysML Security Requirements with Security Observers

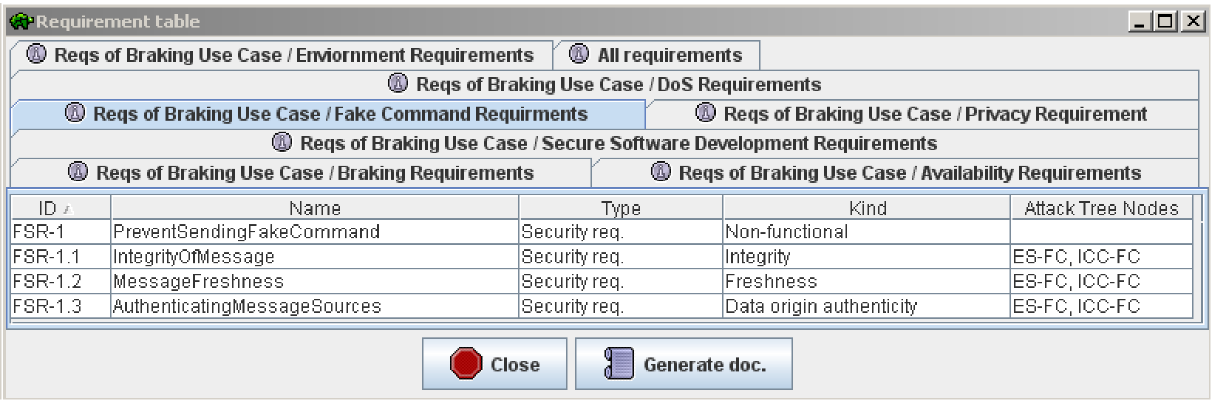
Observers may additionally be seen as a means to document requirements. This set of requirements and observers altogether provides a conceptual model of the security expectations of the system, abstracted from the literary description of use cases. Requirements testing based on the definition of the observers corresponds to the verification of software or system design security patterns and their enforcement, i.e., verification that components or traffic are properly authenticated, that rules regarding data producers are correctly enforced, etc. Those rules were essentially extracted from the use case specifications. In order to trace security requirements for a system we considered several requirement testing and validation approaches [12]. However, due to the EMVY implementation and design specification we are limited to employ only a restricted set of these approaches. Currently, this is achieved mainly through the manual code inspection approach. Based on our analysis, we identified the following security requirements (see Figure 7) which are enforced or partially considered during implementation phase.

- *Controlled Access Control:* Standard protocols and security policies are used whenever appropriate. A controlled access property is enforced to a set of actions and/or information and a set of authorized entities. The property guaranteed that the specified entities are the only entities that can perform the actions or access the information. The property is further detailed with other constraints (i.e., session duration, login interfaces, etc.) on the period of authorization. Controlled access ensures that EMVY entities only have access to information and functions that they are authorized to access as appropriate to their expected activities.

- *Integrity of Messages:* It is important to protect sensitive information as it is being passed from a remote client to a remote server and back. In the current EMVY implementation stack, message integrity is enforced by either using signatures or message authentication code mechanisms. Given the fact that basic security can already protect most applications and that bandwidth is a scarce resource on automotive buses, we decided to allow MAC truncation. i.e. use of only fractions of a calculated MAC. According to NIST and FIPS recommendations cryptographic authentication codes should have a minimal length of 64 bits, when no additional measures to limit the validation rate are taken.

- *Message Freshness:* A message freshness requirement is partially enforced by the EMVY. The property is satisfied during random key generation by the HSM which implicates the key validity period and includes time stamp during session key creation. This session key is used for secure communication between EMVY entities which implicitly indicates that messages are fresh in a given session.

- *Authentication Message Sources:* Authenticity is considered in a multi-step fashion. First, during establishing secure channel between EMVY entities are authenticated using EAM (the required messages are exchanged using the yet unprotected channel offered by the CCM network stack). Upon successful authentication, an EMVY authentication ticket is issued to the entity, the channel is added to the active channels and messages may be exchanged securely. In the later steps, this authentication ticket is used for upper layer authentications. For instance, the need to authorize operations based on RPCs from EMVY/EVITA together with the fact that only channels, not RPC messages are authenticated. In particular, the expression of low level authentication is linked with specific authorization at upper layers.


Requirement table
Reqs of Braking Use Case / Enviornment Requirements
All requirements
Reqs of Braking Use Case / DoS Requirements
Reqs of Braking Use Case / Fake Command Requirments
Reqs of Braking Use Case / Privacy Requirement
Reqs of Braking Use Case / Secure Software Development Requirements
Reqs of Braking Use Case / Braking Requirements
Reqs of Braking Use Case / Availability Requirements
Close
Generate doc.

ID	Name	Type	Kind	Attack Tree Nodes
FSR-1	PreventSendingFakeCommand	Security req.	Non-functional	
FSR-1.1	IntegrityOfMessage	Security req.	Integrity	ES-FC, ICC-FC
FSR-1.2	MessageFreshness	Security req.	Freshness	ES-FC, ICC-FC
FSR-1.3	AuthenticatingMessageSources	Security req.	Data origin authenticity	ES-FC, ICC-FC

Figure 7 Active Brake Security requirements

The results of the requirement validation is collated over all of the EMVY implementation stack, with respect to active brake implementation, that are considered and summarized in terms of security mechanisms. In particular, requirement validation on code provides us a way to build the relationship with abstract security requirements, security building blocks, and enforcement of security mechanisms.

4.5 Dynamic Tests: Intrusion Detection

Dynamic testing has been based on the introduction of a reference monitor logging abnormal events or behaviors coming from an application or an ECU and that may constitute the signature of an intrusion. Dynamically testing the overall system against intrusions is important to address denial of service attacks in particular, as well as making the system more robust against runtime attacks that might result from uncaught vulnerabilities in the implementation or even design weaknesses. The detection of any abnormal behavior may then enable the system to put itself into a failsafe mode restricting its communication capabilities but protecting the normal operation of safety-critical onboard systems. In this respect the Security Watchdog Module (SWD) complements the cryptography based protection mechanisms.

We approached this problem through the introduction of specific support for distributed probing and the development of a centralized logging facility into the EMVY/EVITA communication framework. We also developed specific event filters that make it possible to assess the overall behavior of different buses of the onboard system.

The Security Watchdog (SWD) is an intrusion detection component, that is deployed in a multi-centered and distributed fashion. This means that one or more EMVY-Master nodes can receive data from several different EMVY-client instances.

The SWD features a pluggable interface in order to react on events. With this interface, a plugin may subscribe to certain event types and evaluate the data at its sole discretion. This means that the interface is not limited to basic signature-based or behavior-based checks, but allows all kinds of abstract action and filter classes in order to foresee any kind of input data. We have validated our concept on behavior based examples. As of now, signature based detection is not yet relevant since known attack patterns do not exist in the automotive domain.

4.5.1 Architecture

The security watchdog is monitoring the system via distributed probes or sensors that may report system-intrinsic or environment events. The sensors should be able to send events to the central watchdog instance via EVITA communication to assure the authenticity, integrity, and confidentiality of the messages. The central watchdog gathers these events in a log and notifies registered listeners about newly-received events. Event listeners may themselves generate new events, or generate other actions in the system. For example, the watchdog may change the network policy from a more restricted one to a looser one if the number of sockets connected is constantly high, thus avoiding a denial of resources, while at the same time generating a warning message for the user. Or it may limit communication to only entities crucial for the car's operation, effectively preventing a denial of service.

The EMVY Framework The EMVY framework is written in C (emvylib-remote) and C++ (EMVY-master). The C part is implemented in a library, so that all sensors and actors of the system can make use of EVITA/EMVY functions in order to send probe data, or take appropriate reaction on detected attacks. For us, the most important part of the infrastructure is the communication stack, that currently implements a TCP/IP communication module and a CAN interface based on socket communication. It provides a facility to reliably send a data buffer and receive a response over a connection that can be encrypted and authenticated. This connection is used for the SWD communication.

EMVY's Remote Procedure Call architecture On the EMVY-Master control unit in the vehicle architecture, the entry point to transform the SWD calls the singleton accessor to the SWD. Based on a configuration file flag, either a stub singleton which forwards all event notices to the actual SWD instance, or the actual SWD implementation is returned. The stub is responsible for opening a secure communication channel upon its creation, thus implementing the multi-centered approach, and sending the given security event notices over this channel to a server. The server needs to be implemented as a special EMVY module and must accept connections to an EVITA server port on the socket address specified in the configuration file.

In addition, EMVY clients will communicate to the SWD interface of one of the EVITA master servers through the RPC interface (as described in [16, 15]). The clients may act as probes, supplying the SWD with information, or as actors, that will take action on certain pre-aggregated security events.

4.5.2 Usage and Integration in EMVY

In EMVY, the so-called `SecurityEventNotice` provides a data structure in order to transfer probe data among the SWD system. As an abstract data class, it only provides data fields for

- The `SecurityEventType`: such data describes the type of probe data (an enumeration)
- A timestamp
- An issuing entity identifier, including address information
- A description field.

This means that the description string will be used as a container for SWD payload, so that all values that should be contained in this probe message need to be serialized into a string before they can be sent. This can be done either by translating values into strings by hand, or by means of some serialization framework. For the sake of simplicity we chose to concentrate on encoding single values in a string, which is done by the `Serializer` methods of each class, that are evaluated by the `SerializationManager` when needed. The receiving SWD then deserializes the events through the `SerializationManager` and the appropriate deserialize-methods. It extracts the values through the `DataAccessor` methods, and passes them on to the filter. The filter itself can then evaluate or pre-aggregate data for further processing and/or taking actions.

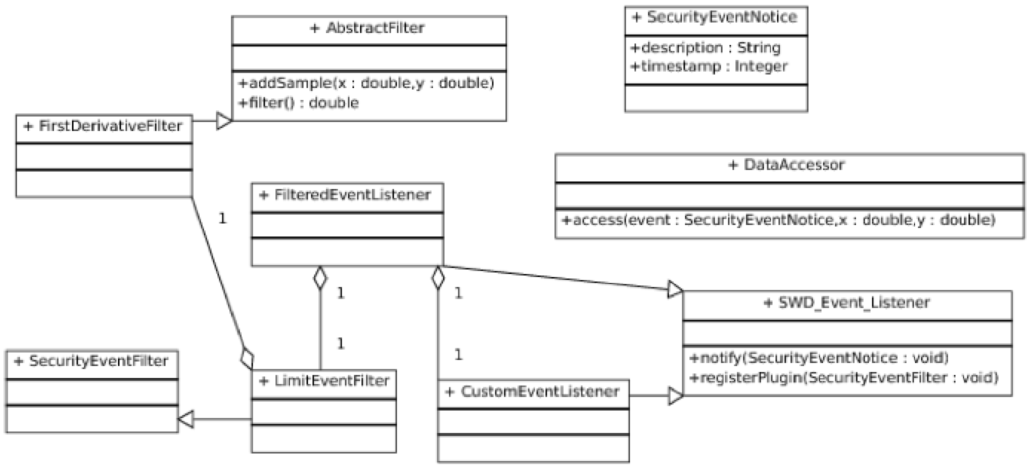


Figure 8 UML Architecture of Generic Filter.

4.6 Proof of concept

In order to show the effectiveness of the integration of intrusion detection into the framework, we have implemented a proof of concept filter and probe application that will take reaction on an atypically increasing number of network connections.

4.6.1 The Probes

A specialized probe-collector has been crafted in order to obtain the network connection number and type from Unix-compatible systems that served as proof of concept (tested on Linux, MacOS X, cygwin-windows). The number of currently open TCP connections is serialized and sent upstream to the SWD framework. A sample is taken every second.

4.6.2 The Filters

In contrast to the probe, we have crafted the filter in a way that it can be applied to multiple types of automotive sensor data. For example, stream processing using a timed window is a common approach in order to detect malfunction of vehicle components: for instance, such systems rely on rules like "if the gasoline level of the tank drops by more than 20% within half an hour, one should raise an alert".

Consequently, we have designed a generic derivate filter as well as a classic limiting event filter: the `LimitEventFilter` can be parameterized with minimum and maximum allowed values (also commonly used for vehicle diagnosis: for instance, cooling-water temperature shall not exceed 120° C.). To allow a more fine-grained decision, the `FirstDerivateFilter` uses the actual differences between measurements and can be parameterized with a maximum (negative or positive) gradient (e.g., if the temperature rises by more than 20° C in one minute).

We have implemented these filters with some of the intrusion detection techniques described by Müter et al. [10] in mind. These authors categorized possible intrusions

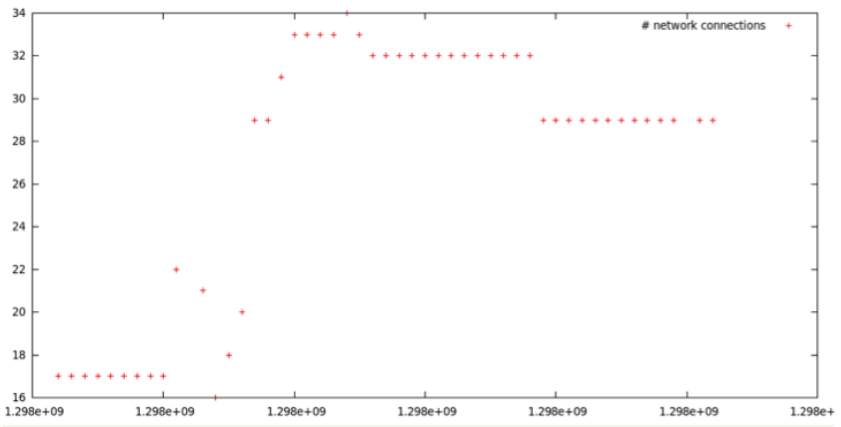


Figure 9 Absolute amount of open network (TCP) connections at a time. Samples are taken every second.

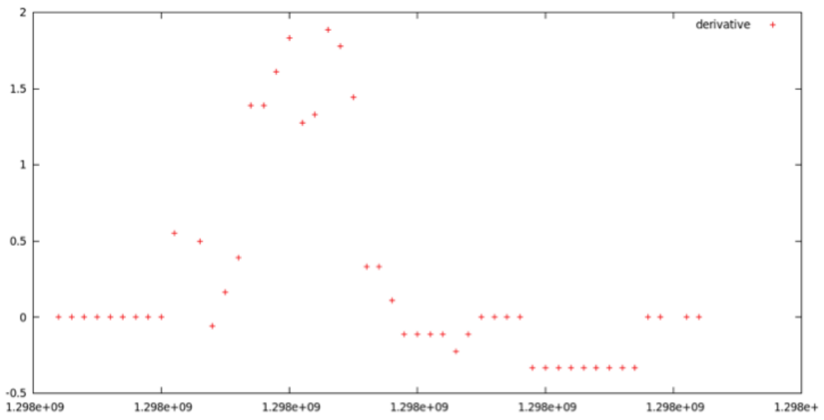


Figure 10 Relative amount of opening and closing network (TCP) connections per second (first derivative). The time between adjunct data points is one second.

detection layers as packet-level, network-level, and application level. Our sensor can be used to generate probe data for all these layers. Our example in the following section can be applied to the following types as defined by Müter et al.: Formality, Compliance, Frequency.

4.6.3 Experimenta Data

We have recorded data with a Probe and the FirstDerivate filter in place. You can see the number of open TCP connections on the computer in Figure 9. The corresponding derivate plot is shown in Figure 10. You can clearly see how an absolute and a derivate filter can be engaged at certain thresholds, e.g., the amount of total open connections could be limited at 25 (LimitEventFilter) or at 1.5 additional connections per second (FirstDerivateFilter).

5 Conclusions and Future Work

We discussed the results of the security-minded tests and code validations that were applied to the EVITA system, in particular during the software development of the EVITA framework. In this respect, our results were rather useful since we found several potential vulnerabilities, some of which were corrected. The tests and validations run after software development highlight some potential areas of improvement, mostly in terms of safety (thread synchronization most notably) on some components that should be seriously reconsidered for a commercial exploitation of the existing code base. We also described our approaches to testing on a vehicular on-board system comprising mutiple ECUs, both statically and dynamically. We believe that such approaches might be successfully reproduced in the framework of other similar systems. We are still in the process of developing tools to automate model-based tests and static analyses, as well as to instrument binary code for testing, in particular with respect to the integration of multiple components but their design is still clearly an open research problem.

We should finally emphasize the fact that the EVITA software framework and demonstrator - which we are validating - is by no means a commercial and polished product but instead a research prototype that cannot be directly integrated into production vehicles in its current state. Additional code validations - e.g. using formal verification or testing techniques - would need to be performed when the HSM related software or EMVY framework are integrated with a particular industrial bus. Vulnerabilities should of course be sought in the final implementation - in which interactions between previously independent software components might then occur.

Disclaimer: *We performed tests of individual EVITA components and —to some extent— of the system design and integration. Despite this, for every deployment target, additional adequate security measures should be taken (e.g., non-executable and randomized stack configurations for x86 kernels). The security of such deployment targets must be individually assessed by security analyses and penetration tests. While this is beyond this document's scope and only necessary for industrial deployment, we would like to stress the fact, that such analyses are equally important to the soundness of the EVITA system itself and essential for system security.*

References

[1] EB-tresos, ECU Software Development. http://www.eb-tresos-blog.com/.

[2] ELDK, Software for Embedded Linux systems. http://www.denx.de/.

[3] The HiTOP IDE for Infineon Tricore by Hitex. http://www.hitex.com/.

[4] Tricore Software Development Tool by Altium. http://www.tasking.com/.

[5] By Anton Cervin, Dan Henriksson, Bo Lincoln, Johan Eker, and Karl-erik Å rzén. Analysis and Simulation of Timing. *IEEE Control Systems Magazine*, (June):16–30, 2003.

[6] C. Fischer, F. Pirklbauer, and M. Mittendorfer-Holzer. HSM Low Level Driver Specification. Technical Report Deliverable D4.2.2, EVITA Project, 2011.

[7] T. Gendrullis, M. Wolf, and H. Platzdasch. Hardware Implementation Specification. Technical Report Deliverable D4.1.1, EVITA Project, 2011.

[8] Vincent C. Hu, Evan Martin, JeeHyun Hwang, and Tao Xie. Conformance checking of access control policies specified in xacml. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 02*, COMPSAC '07, pages 275–280, Washington, DC, USA, 2007. IEEE Computer Society.

[9] M.S. Idrees, Yves Roudier, and Ludovic Apvrille. A Framework Towards the Efficient Identification and Modeling of Security Requirements. *5th Conf. on Network Architectures and Information Systems Security*, 2010.

[10] Michael Müter and André Groll. Attack detection for in-vehicle networks. In *VDI Conference on Automotive Security*, 2009.

[11] Gabriel Pedroza and Ludovic Apvrille. LLD Modeling, Verification and Automatic C-Code Generation. Technical Report Deliverable D4.2.3, EVITA Project, 2011.

[12] Y. Roudier, H. Schweppe, and L. Apvrille. Test Specification. Technical Report Deliverable D4.4.1, EVITA Project, 2011.

[13] A. Ruddle and et al. Security Requirements for Automotive On-board Networks Based on Dark-side Scenarios. Technical Report Deliverable D2.3, EVITA Project, 2009.

[14] H. Schweppe, M. S. Idrees, Y. Roudier, B. Weyl, R. El Khayari, O. Henniger, D. Scheuermann, G. Pedroza, L. Apvrille, H. Seudié, H. Platzdasch, and M. Sall. Secure On-Bard Protocols Specification. Technical Report Deliverable D3.3, EVITA Project, 2010.

[15] H. Seudié, E. Akcabelen, I. Ipli, H. Schweppe, Y. Roudier, and S. Idrees. Second version of the implementation of the software framework. Technical Report Deliverable D4.3.2, EVITA Project, 2011.

[16] B. Weyl, H. Seudié, J. Shokrollahi, B. Weyl, A. Keil, M. Wolf, F. Zweers, T. Gendrullis, M. S. Idrees, Y. Roudier, H. Schweppe, H. Platzdasch, R. El Khayari, O. Henniger, D. Scheuermann, L. Apvrille, and G. Pedroza. Secure On-Board Architecture Specification. Technical Report Deliverable D3.2, EVITA Project, 2010.