Project acronym:	OVERSEE
Project title:	Open Vehicular Secure Platform
Project ID:	248333
Call ID:	FP7-ICT-2009-4
Programme:	7th Framework Programme for Research and Technological Development
Objective:	ICT-2009.6.1: ICT for Safety and Energy Efficiency in Mobility
Contract type:	Collaborative project
Duration:	01-01-2010 to 30-06-2012 (30 months)

Deliverable D2.5:

Definition of the Requirements for Validation of OVERSEE Implementations

Authors:	Cyril Grepet (Trialog) Alfons Crespo (UPV) Nicholas McGuire (OpenTech) Thomas Enderle (Escrypt)
Reviewers:	Alfons Crespo (UPV) Nicholas McGuire (OpenTech) Jan Holle (UniSi) André Groll (Unisi)

Dissemination level:	Public
Deliverable type:	Report
Version:	1.3
Submission date:	11 November 2013

Abstract

This document contains an overview of the technical requirements to provide a validation of the OVERSEE platform. In the context of the project it means to define a set of methodologies, techniques and tools suitable to provide a sufficient validation of the various part of the platform to be compliant to some standard in security (and safety) as well to ensure that an implementation fits the OVERSEE platform specification. Each one can have consequences on the design of the platform.

Contents

List of Figures

List of Tables

List of Acronyms and Abbreviations

API	Application Programming Interface
ATC	Abstract Test Case
ATS	Abstract Test Suite
BSI	Bundesamt für Sicherheit in der Informationstechnik
CC	Common Criteria
CPU	Central Processing Unit
ETS	Executable Test Suite
ETSI	European Committee for Standardization
FLOSS	Free/Libre Open Source Software
FSM	Finite State Machine
HSM	Hardware Security Module
I/O	Input Output
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
ITS	Intelligent Transportation System
IUT	Implementation Under Test
LT	Lower Tester
MAF	Major Frame
MMU	Memory Management Unit
MTS	Methods for Testing and Specification
NSA	Natinal Security Agency
OS	Operating System
OVERSEE	Open Vehicular Secure Platform
PCO	Point of Control and Observation
PP	Protection Profile
RCS	Revision Control System
SAR	security assurance requirements
SFR	security functional requirements
SSPM	Security Services Provider Module
ST	Security Target
SUT	System Under Test

TCP	test coordination procedure
TOE	Target Of Evaluation
TTCN	Testing and Test Control Notation
UT	Upper Tester
XML	Extensible Markup Language

Document History

Version	Date	Changes
1.3	26-01-2012	

1 Introduction

The Open Vehicular Secure Platform (OVERSEE] project has produced this deliverable. It contains contribution from TRIALOG, escrypt GmBh, UPValencia and OpenTech.

1.1 Scope and Objectives of the Document

The scope of this document is to describes a set of methodologies based on well-known techniques and tools, to validate the different part of the OVERSEE platform.

There are a number of methodologies to validate a product depending on its scope. Formal, semi-formal or practical conformance testing for communicating systems and verification by proof or model-checking for critical systems are examples of this diversity.

In this document is presented some methodologies suited to validate the OVERSEE platform in its communication, security and isolation aspects.

1.2 Document Outline

The section 2 is a short overview of the state of the art in the three methodologies that will be used in the following of the project.

The section 3 presents how to validate the communicating part of the system by conformance validation. The section 4 presents the application of Common Criteria to a specific subset of security objectives. Finally, a verification using Model-Checking of the isolation aspect of the OVERSEE platform is described in section 5.

2 State of the art: Overview of Validation possibilities

2.1 Conformance Testing

This section assesses the use of typical communicating system testing approaches based on the ISO-9646 Standard and adapts it to the testing of certain aspects of the OVERSEE platform. The ISO standard is used as the foundation of the ETSI ETS-300.406 Standard.

The purpose of conformance testing is to determine if system's behaviour meets its specification. The specification could be literal as in most of the standard or based on a formal/semi-formal specification. The ISO-9646 standard defines the conformance testing for the implementation of protocols and therefore all communicating systems that use message-based communication. The conformance testing is one of the approaches to ensure the conformance of an implementation with respect to its definition but also to ensure the interoperability between multiple implementations.

One of the main benefits of this kind of validation is that it can be applied to an implementation that code is not reachable. It is called *Black Box testing* by opposition to the *White Box testing* where the code is available to the test.

Conformance testing should be used to validate both communication protocols and interfaces. Section 3 provides more information and describes the possible uses of Conformance Testing within the OVERSEE project.

2.2 Common Criteria

The Common Criteria (CC) is a collection of rules to formalize security evaluations. They were written in a joint effort by the government agencies responsible for IT security in various countries (e.g., NSA for the US, BSI for Germany etc.) These parties have a mutual agreement to recognize CC evaluation results.

The CC first defines a model and a language in which security problems and solutions are described, and subsequently specifies a process how to evaluate a product depending on the required security level. Not only the implementation is evaluated, but also associated issues like development, testing, distribution etc. The evaluation procedure is very formal and modular which ensures the applicability of the method to a wide range of IT security problems while retaining comparability of security properties and evaluation results.

The following roles are defined: the developer, the evaluator, the sponsor and the authority (commonly one of said government agencies). The role "customer" is indirectly involved. The sponsor arranges an evaluation of a product created by the developer to convince the customer that the product is secure to a certain degree. To this end the sponsor employs an evaluator, who is accredited by the authority and is trusted by the customer.

The end result is a widely recognized document which assures certain security properties of the product.

2.3 Verification of properties by Proof and Model Checking

Over the last years, formal methods have become more and more important to assure the quality of software, and to complement traditional testing methods. The main problem of traditional testing methods is, that even for relatively simple applications, they can never be exhaustive, so formal methods are getting a more and more important part of quality assurance processes.

The detailed evaluation of what methods can be smoothly integrated into the actual development process for OVERSEE, respectively which would be integrated in the form of proof-of-concept is still open as development has not yet fully started. Formal methods are seen as one strong catalogue of methods that can help resolve some of the open issues related to reusing existing components while operating in an environment that has not only security but also some limited safety and obvious availability demands. Based on the practical evaluation of existing tools a set will be proposed for integration into the OVERSEE build-environment

3 Conformance and Interoperability testing (Trialog)

3.1 Introduction

As stated in section 2.1 testing methodology is based on international standards. This work relies on the following ones:

- ETS 300 406: Methods for Testing and Specification (MTS) - Protocol and Profile Conformance Testing Specifications - Standardization Methodology [1] .
- ETR 266: Methods for Testing Specification (MTS) – Test Purpose Style Guide [2].
- ISO/IEC 9646-x Information Technology - Open Systems Interconnection - Conformance Testing Methodology and Framework [3].

The ETR 266 is not a standard but rather an informative document nevertheless it is still useful.

3.1.1 Definitions

Some definitions come with the standard.

- **ATC**: Abstract Test Case – A realisation of a test purpose. It is a complete specification of all the actions required to achieve this test purpose. It is written in a programming language specific to testing, e.g., TTCN. In general, each test case starts in a stable testing state and ends in a stable testing state
- **ATS**: Abstract Test Suite – An abstract test suit is a collection of test cases that are used to test the Implementation Under Test to show that it correctly executes the specified set of behaviours. A test suite can include detailed instructions or goals for each collection of test cases as well as information on the system configuration to be used during testing. A test or group of tests may also contain prerequisite states or steps, and descriptions of resulting states or steps.
- **ETS**: Executable Test Suite – This test suite can be executed by a program that can communicate with the System Under Test. Usually the ETS is automatically generated from the ATS.
- **IUT**: Implementation Under Test – This is the particular portion of the System Under Test which is to be tested, e.g., the communication functions of a mobile phone but not the sound or screen functions. The IUT is viewed as a black box by the test system is compared to its specification
- **LT/UT**: Lower Tester / Upper Tester – TTCN was first specified for protocol testing in the telecommunication domain. Therefore LT and UT refer to the layers of the protocol stack. Lower Tester: the test component of the test engine that communicates with the IUT via the Points of Control and Observation at the lower layer. Upper Tester: the test component of the test engine that communicates with the IUT via the Points of Control and Observation at the upper layer

- **PCO**: Point of Control and Observation – Communication between the test engine and the IUT is achieved via points of control and observation. PCOs are the only means that allow the test system to interact with the IUT. At least one PCO is needed in a testing architecture.
- **SUT**: System Under Test – defines the boundaries for the system that is being tested for correct operation, including the hardware and software, e.g., the complete hardware and software of a mobile telephone is the SUT when the IUT is the communication functions of the phone. Basically the system that contains the IUT. The IUT and the SUT can be the same in many cases($SUT \sim IUT$). Sometime the interfaces of the IUT are not directly available therefore the IUT is imbricate in the SUT ($SUT \supset IUT$)
- **Test Case:** A set of conditions or variables under which a test will determine whether an application or software system is working correctly or not.
- **Test Purpose**: A prose description of a well-defined objective of testing, focusing on a single conformance requirement. It represents an abstract description of the test to be performed independently of any concrete realization.
- **Test Report**: the result of each test generated by the Test System to provide data to evaluate the IUT's conformance to the requirements
- **Test System (or Test Engine):** includes the dedicated test system and the tests that it executes. It has full control over the SUT while stimulating it with relevant events and checking its resulting behaviour.
- **Testing and Test Control Notation (TTCN)** [4]**:** a programming language standardised by ISO for the specification of tests for real-time and communicating systems, developed within the framework of standardised conformance testing (ISO/IEC 9646 [3]). A TTCN test suite consists of multiple test cases written in the TTCN programming language. Dedicated compilers or interpreters are required for execution.

3.1.2 Conformance Validation Process

The process usually used to validate a protocol or a communicating system is usually the one described in Figure 1.

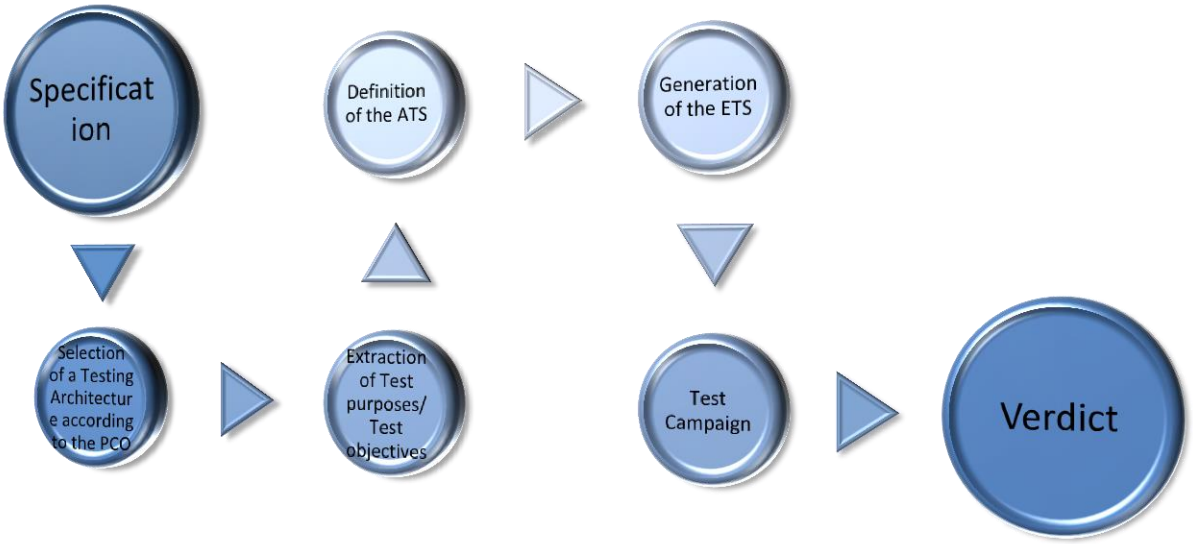

Figure 1: Conformance testing process

The steps are the following:

- According to the SUT and the available PCO a testing architecture is define enabling the test campaign. A test architecture is usually compose of some *Testers* connected to the PCO of the SUT that are able to communicate with the IUT
- From the specification some test purposes are extracted. The test purposes can have different objectives: to capture the technical description of the specification (e.g., the required functionalities), to test the boundary conditions and divergence of the specification (e.g. if a parameter can take two value ranges that lead to different behaviour problems usually occur at the boundaries), to test internal structure not accessible (e.g. the correct filling of a routing table). Depending on the format of the specification it can be manually (if it is a textual one) or automatically (in case of formal specification)
- The test purposes serve to define various test scenarios in a dedicated format (usually the standard TTCN-3). Each test scenario describes a succession of actions that verify if the IUT react correctly to a precise sequence that fulfil one goal (i.e. test purpose). The whole test scenarios are the Abstract Test Suite (ATS). An ATS is usually readable and understandable.
- From the Abstract Test Suite an Executable Test Suite (ETS) is generated. It is the translation of each test scenario from the TTCN-3 format to an executable code that will stimulate the real IUT through the *Testers*.
- The test campaign consists in executing the ETS through the tester and to collect all the output from the IUT. If each input lead to an expected output then the verdict "Pass" is provided. That means that no error has been exhibited (and not that the implementation is without fault). If some output are not excepted the verdict is

therefore “Fail”. From the error detected, the implementation must be corrected and a new test campaign will be played later.

With quite the same process interoperability testing could be performed between more than one implementation.

Testing architectures are also defined in [1] [3]. The testing architecture is usually composed for protocol testing of:

- An Upper Tester (UT) linked to a PCO between the IUT and the upper layers of the system.
- A Lower Tester (LT) linked to a PCO between the IUT and the lower layers of the system.
- A Test Engine to coordinate both testers and to evaluate the conformity of the test sequences. The coordination is usually unsure by a test coordination procedure (TCP)

A representation of a local architecture (as define in [3]) within the conformance testing process is depicted in Figure 2

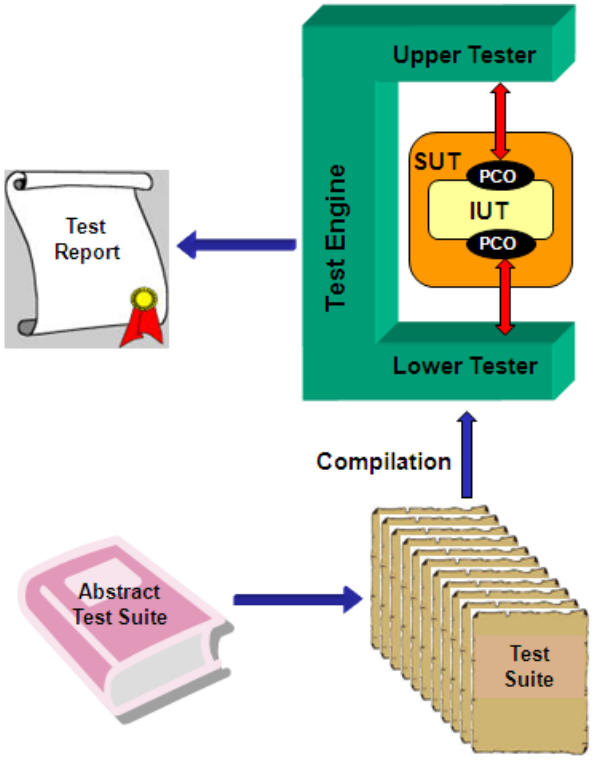

Figure 2: Local Testing Architecture and Conformance Testing Process

Interoperability is usually defined as following: Interoperability testing is the activity of proving that end-to-end functionality between (at least) two communicating systems is as required by those systems' base standards. EG 202 237 describes a generic approach to interoperability testing.

The interoperability testing relies on the same concept and phases even if their name can be slightly different.

3.2 Conformance testing in OVERSEE

3.2.1 Goal of the Conformance Testing

As the OVERSEE platform will support various communication protocols as well as multiple applications (mostly dedicated to enforces security and dependability in the system) it will be necessary to use the conformance testing

- Conformance of protocol implementations: In order to ensure separation of concerns, there could be different test systems focusing on systems under test (SUT), i.e. different layers.
- Conformance of interfaces: In order to ensure separation of concerns, there could be different test systems focusing on different interfaces.

Even if the two goals are different the techniques are the same. As the design of the OVERSEE platform, the involved communication, the security services and most of the components are designed in the same time as this deliverable (see deliverables D2.1, D2.2, D2.3 and D2.4) it is not possible to drawn a precise picture of the testing architecture or the position of the PCOs. These concerns will be addressed in the coming WP4: Open Platform Validation Support.

Within the OVERSEE project many protocols) are considered. Some of them are not yet well-known as the ITS 5.9 GHz-based one. It will be necessary during the project, and for any development of such a platform to ensure the conformity of the implementation of the protocol (if it is not an already existing one) and the interoperability between the protocol itself and the interfaces of the platform.

In a wider scope, any application running on top of the hypervisor layer is to be tested as interoperable with the platform itself. For instance, if a security service is implemented in the Security Services Partition, it has to be tested as:

1. Conform to its specification
2. Interoperable with the platform

Limitation of the real implementation: A complete conformance and interoperability testing rely on a detailed and non-ambiguous specification of the system. Since OVERSEE aims to prove only the suitability and the feasibility of its approach to a real implementation, a complete conformance and interoperability testing seems not possible to perform along the project. Nevertheless, it is still possible to conduct a series of tests based on non-formal specific purposes to cope with the conformity and the interoperability of the various protocols and interfaces.

3.2.2 High Level requirements for testability

From the purpose of the Conformance testing as a validation methodology for the OVERSEE platform some technical requirements can be easily derived.

In the Table 3-1 high level requirements are described by a categorization on their goal.

Req	Global Requirements	SubReq	Requirement
1	Monitoring of the communication interface by the Test Tool	1.1	*Platform provides Points of Control and Observation (PCOs) on each communication interface*
		1.2	*The test tool must be able to plug on all the interfaces provided by the platform*
2	Observability of the interfaces between the building blocks by the test tool	2.1	*Each building block must be reachable i.e. "communication interface" even if communication is not really the good term*
		2.3	*The test tool must be as fast as the building block (to avoid message congestion reaching the test tools)*
		2.4	*The test tool must support the protocol used by the platform*
		2.5	*The test tool must support and be coherent with the cryptographic, signature and keys used by the platform (can be divided in various requirements)*
3	Reset ability of the system to the "initial state"		*Each building block must allow one to drive them in a known state, permitting therefore the test*
4	Testability of non functional requirements (Observability of resources consumption by the test tool)		*The platform must provide an access to each resource to be tested.*
5	Well-defined of API and Data of building blocks		Each building block must specify clearly its API and the data exchanged (I/O)
6	Responsivity of command		Each building block must be able to provide a sort of acknowledgment that each command received has been performed (i.e. always an Output for an Input)

Table 3-1: High Level Requirements Table

Security aspect:

It is necessary to highlight that these requirements imply that only an authorized actor can be able to perform the test for security reasons.

For instance the requirement 1.1 could be described as: Each communication interface in the OVERSEE platform have to be reachable for an external and authorized user to allow verification and validation. It means that some PCOs must been reachable for an authorized tester. This requirement will allow one to perform a wide range of functional validation. If anyone can use the PCO it could have a great negative impact on the system.

4 Security Services Validation

4.1 General CC Process

The product or system that is the subject of the evaluation is called TOE (Target of Evaluation) [5].

In a first step the developer prepares a ST (Security Target) about the TOE. This is a document consisting of:

- introduction
- conformance claim
- security problem definition
- security objectives
- extended components definition
- security requirements
- TOE summary specification

The security requirements in turn are composed of

- **SFR** (security functional requirements)

  SFRs are formalized specifications of the security functions, consisting of predefined functions [6] and additional parts defined by the developer in the section “extended components definition”

  The developer models the security functions of the TOE in terms of these defined SFRs.
- **SAR** (security assurance requirements)

  SAR’s specify how the TOE should be developed, implemented and evaluated. SAR’s are divided into the following classes:
  - Development
  - Documentation
  - Life Cycle Support
  - Security Target Evaluation
  - Tests
  - Vulnerability Assessment

  Every SAR specifies the actions to be done in the evaluation process by the developer on the one hand and by the evaluator on the other hand, and as well the documents to be produced.

  All possible SARs are pre-specified in [7]. Many of them are available in different hierarchical graded variants. Based on the desired assurance level an adapted subset is selected for an ST.

An Evaluator then carries through the actions described in the SARs and assures that the Developer has fulfilled all the requirements made there. This is described in [7], in [8] there is a more detailed explanation of this process. The TOE has to fulfill every criterion for a

successful evaluation (or a version thereof that was sanitized of proprietary information).In the final stage an evaluation report together with the ST is published.

It is important to notice that the CC do not require certain security properties by itself. The disadvantage of this approach is that a customer cannot rely blindly on a “certificate”, but has to compare the ST to his requirements first. The advantage of this approach is that it is very flexible and applicable to a wide range of IT products.

4.2 Application to OVERSEE

4.2.1 Introduction

Our implementation of OVERSEE will be a Proof-of-Concept. At a later point in time production implementations will be created. The goal is to support developers of those implementations with the evaluation of their product. Currently the evaluation approach in the Common Criteria is not tailored to the automotive area. In WP4 we will therefore adapt the Common Criteria approach and apply it to our design and proof of concept implementation. Consequentially we will abbreviate steps that are too specific to our implementation and as a result would not be helpful to the developers for the evaluation of actual implementations.

TOE will be composed of:

- Security Services Partition
- Security Services Provider Module (SSPM)
- Hardware Security Module (HSM)

XtratuM itself as well as the HSM itself will be excluded from the evaluation, which means:

- The HSM is assumed to be secure.
- The communication between the Security Partition and the HSM is assumed to be secure and unreachable by an attacker.
- Communication between the SSPM and the Security Services Partition is assumed to be secured by XtratuM, which means it cannot be attacked from other partitions that are not involved.
- The isolation provided by XtratuM is assumed secure.
- As a result the only interfaces that are considered exposed to attacks are the connection between a user partition and the Security Services Partition, and, to a limited degree, the API of the SSPM.

4.2.2 Evaluation Support

We will now give an overview of the process and decide where we will support the developer of a productive implementation.

4.2.3 Development

Most of the requirements in this section are general enough to be useful to support evaluation of actual implementations. They result mainly in an extensive documentation.

- Security architecture description (ADV_ARC)
  Here the developer first has to ensure, that the security functions cannot be bypassed, and that the TOE protects itself from tampering. Then he has to describe the security architecture of the TOE.
- A functional specification of the security functions has to be given. (ADV_FSP)
- The Implementation representation (i.e. the source code) has to be provided (ADV_IMP)
- The design has to be described. (ADV_TDS)

4.2.3.1 Guidance Documents

The CC requires a complete, detailed user guide for all kinds of persons using the external interfaces of the TOE, including developers, administrators and normal users. We will only have a rudimentary user interface for demonstration purpose, so there is no use in extensively documenting the interface for users and administrators, but we will document the API for developers.

Also the description of preparative procedures lies outside of our scope.

4.2.3.2 Life Cycle Support

Our implementation will be a prototype, so this section doesn't apply. Nevertheless by using sound development practices one already fulfills a larger part of the requirements in this section, e.g., version control systems, build systems, bug tracking, coding standards etc…

4.2.3.3 Security Target Evaluation

In this part the exact requirements for the ST are given. As the ST is a document on a rather high level, it should be applicable to most implementations of the security services in OVERSEE with minor changes.
We will in general adhere to the most recent CC version, this is version 3.1.
This is also the place to include Protection Profiles (PP) that are to be fulfilled. A PP is structurally similar to an ST, but it deals with abstract classes of TOE, instead with concrete TOE. There are some predefined PP available on [TBD]. An ST author can then claim conformance to (or "include") as many of these PP as he like and then has to fulfill all of them. This is a practical way to facilitate ST creation, but unfortunately there is no PP available that is usable for our purpose.
The predefined SFR will likely not be sufficient in this automotive application, so we will have to define some extensions.

4.2.3.4 Tests

Here the security functions of the TOE are to be tested. This is a special case of testing, because the focus is on security. An analysis of the coverage and the depth of the supplied tests will also be done.

All material generated in this task should be directly reusable by developers of future implementations of OVERSEE, as long as APIs are concerned.

4.2.3.5 Vulnerability Assessment

The evaluator has to conduct a vulnerability assessment, based on a predefined model of an attacker. Unfortunately this process strongly depends on the actual implementation; hence if at all we will only be able to give hints to where vulnerabilities may occur.

5 Verification of Isolation Property

5.1 State of the art of the techniques

5.1.1 General introduction to the core problem

Complex code has a very large state space, making exhaustive testing practically impossible. Think of a trivial function to add two integers.

*int add(int *a, int *b)*

The total input space of this function, for a 32 bit system, is the combinatorial size of the inputs, which would be 2^32 * 2^32 - that is 18446744073709551616 possible inputs - the reason why this can't be exhaustively tested is I guess evident.

So all formal methods are targeting to reduce the state space to a reasonable size allowing to cover as much as necessary to prove specific properties.

State space reduction - a brief summary:

State space reduction can be done by a lot of technologies and many of the model checkers have gained there suitability for large software systems only due to extensive research on state space reduction methods and optimal state space traversal strategies (i.e. SPIN [9]) some very general methods:

- reduce state space based on heuristics:
  - lint → based on heuristics
  - sparse → known problem checker in specific context
- reduce state space to the part related to the bug !
  - satabs → reduction by iterative refinement (CEGAR)
  - stanse → reduction by "Finite State Machine(FSM) extraction"

Many of the tools use a combination of strategies, as this is not relevant at this point we will not go into further details of state space reduction methods.

Aside from the state space problem we also have a serious problem with describing the necessary properties in machine readable form - computers are not too good at understanding language that is easy for humans to comprehend, thus a more formal language is needed. One big show-stopper for formal specification and consequent validation has been the very non-intuitive language constructs used to specify such systems. This has changed to some degree in the past decade. In the following section we exemplify these changes by showing concrete examples of some tools.

5.1.2 Formal specification of critical properties:

The properties of critical calls can be specified in a language suitable for interpretation by bounded model checkers (i.e. CBMC) - allowing to certify that a particular piece of code is satisfying a set of claims that are automatically generated. This is achieved by symbolic execution allowing to cover the complete relevant state-space of the described code. The level of detail of claims can be from relatively simple context checking i.e.:

```
unsigned long __get_free_page(gfp_t gfp_mask)
{
 __CPROVER_HIDE:
    if (gfp_mask & __GFP_WAIT) {
        assert_context_process();
    }
}
```

typical availability problems, like deadlocks, that are hard (or impossible) to detect by traditional testing due to the size of the state-space:

```
void spin_lock(spinlock_t * lock)
{
 __CPROVER_HIDE:

#ifdef DDV_ASSERT_SPINLOCK
    __CPROVER_atomic_begin();
    __CPROVER_assert(lock->init, "Spinlock is not
initialized");
    __CPROVER_atomic_end();
#endif

    do
    {
        __CPROVER_atomic_begin();
        if(lock->locked == 0)
        {
                lock->locked = 1;
                __CPROVER_atomic_end();
                return;
        }
        __CPROVER_atomic_end();
    }
    while(1);
}
```

To assessing the correctness of dependant code, i.e. the initialization and the runtime usage of specific address ranges:

Note that the request call not necessarily is in close proximity to the actual accessed code, further code may well be misinterpreted by humans, i.e. traditional off-by-one array issues or the like. With a simple recording - shown here for the request region call in the Linux kernel and a subsequent test for the use of the proper port value in later calls (below) a set of correctness properties can be derived automatically and then verified with the aforementioned tool CBMC.

```
struct resource *request_region(unsigned long start, unsigned
long len, const char *name)
{
  unsigned int i;
    struct resource *resource = (struct
resource*)malloc(sizeof(struct resource));

    ddv_ioport_request_start = start;
    ddv_ioport_request_len = len;

    return resource;
}


void outb_p(unsigned char byte, unsigned int port)
{
 __CPROVER_HIDE:
    ddv_correct_port_use(port);
}
```

The essence of these seemingly simple examples is the automation, allowing to verify semantic properties of complex code as a routine effort during development rather than trying to test all possible combinatorials - an obviously impossible task for even the most trivial function.

Frama-c/ACSL:

Frama-c is a framework not in itself a tool. It contains a large number of tools to generate basic analysis based on different plug-ins allowing to provide formal specification of simple properties - i.e. locking correctness properties by providing formal specifications in ACSL right in the code.

```
/*@
  requires \valid(ghost_loctable + m);
  requires !ghost_loctable[m];
  ensures ghost_loctable[m];
  assigns ghost_loctable;
 */
void acquire_lock(int m) { ghost_loctable[m]++; }

/*@
  requires \valid(ghost_loctable + m);
  requires ghost_loctable[m]==1;
  ensures !ghost_loctable[m];
  assigns ghost_loctable[..];
 */
void release_lock(int m) { ghost_loctable[m]--; }
```

or providing formal specifications of function prototypes allowing to include them as specification files:

```
/*@ requires \valid(p) && \valid(q);
    ensures *p <= *q;
    ensures (*p == \old(*p) && *q == \old(*q)) ||
               (*p == \old(*q) && *q == \old(*p));
*/
void max_ptr(int* p, int*q);
```

one of the essential properties of formal specification close to the code level is that it allows to provide complete contracts that map to the code and thus allow verifying very specific properties of the actual code. Any change to the code that might only have a side effect in a corner case - and thus evades detectio n by testing is highly probable to be detected provided the specification actually constitutes a complete contract.

```
#include <stdio.h>
int foo(int x) {
    while (x > 0) {
      /* @ breaks x % 11 == 0 && x == \old (x);
         @ continues (x+1) % 11 != 0 && x % 7 == 0 && x ==
\old (x) -1;
         @ returns (\result +2) % 11 != 0 && (\ result +1) % 7
!= 0
         @           && \ r e s u l t % 5 == 0 && \result ==
\old (x) -2;
         @ ensures (x+3) % 11 != 0 && (x+2) % 7 != 0 && (x+1)
% 5 != 0
         @           && x == \old (x) -3;
         @ */
      {
         if (x % 11 == 0)     break ;
         x--;
         if (x % 7 == 0)   continue ;
         x++;
         if (x % 5 == 0)    return x;
         x--;
      }
    }
    return    x;
}
```

practically this provides a form of development level diversity, that is the developer is required to keep code and specification in sync thus making it quite unlikely that a undesired and unintended behaviour would be on the one hand coded and on the other hand specified formally - though of course this is not impossible.

A further advantage of this form of formal specification is the close proximity of the specification and the actual implementation, typically writing lengthy specifications just results in the same being at best ignored in the worst case misinterpreted and the diversion not detected due to the high effort necessary to actually detect such a semantic mismatch (assuming that the compiler is taking care of the syntactic properties sufficiently well).

5.1.3 Tracking tools

Tracking has a few dimensions in complex software, one issue is tracking the sources and the changes, with hopefully meaningful commit messages. A further dimension is tracking movement of code and correlation of code/bugs in a dynamic code base. These properties are not satisfied by most content management system - including the currently in mis-used subversion repository (an aggregation of primarily meaningless commit messages is testimony to how little utility a pure content management system actually is). Proper tools, notably those that have proven the test of time in large software projects like the Linux kernel are:

5.1.3.1 Source tracking with GIT

Probably THE revision control and source tracking tool in the FLOSS world is GIT. Coming out of the need for an open source distributed RCS system, GIT developed very fast, and is now in use at almost all major FLOSS projects. Also repository hosting sites as e.g. github or gitorious reflect the popularity of GIT. The most noticeable advantage of GIT is it's distributed architecture, allowing all possible workflows. Furthermore, the "everything is local" paradigm, not only allows the offline use of GIT, but also makes it very fast compared to traditional centralized repositories (e.g., subversion), making speed a major advantage of GIT.

Apart from that GIT offers lots of possibilities to maintain the traceability of source code. These tools to trace the origin of the source code start at the commit level (git log), down to every single line (git blame). Furthermore tags like signed-off-by or reviewed-by allow the traceability of reviews at the patch level. One bit feature of GIT is the possibility of integrating other tools of your development life cycle via GIT hooks. This could i.e. be used to automatically check the coding style, run some formal tools like static code checkers on every modified source file, ...

5.1.3.2 Context sensitive semantic tracking Herodotos

Detecting bugs is the first step, but bugs tend to move in the code, especially during development, tracking bugs only at the level of the human context (i.e. files, directories) does not do it - they will reappear in different locations, resulting in hours wasted manually re-identifying the culprit. Automatic tracking in database tools is a first step, allowing to determine regressions and maybe prioritize certain problems - there is a plethora of tools to do this (bugzilla, ticket systems, etc)

The fundamental problem with these database focussed solutions though is that all tools result not only in detection of bugs but also in a certain rate of false positives being reported. These then can be cleared by manual inspection... which is obviously not a very efficient way and also lacks effectiveness as we tend to become sloppy if we are ask to re-inspect the same sequence over and over again - most notably in security related systems this can be fatal. What is needed is a tracking system that can correlate moving false positives to ensure that the one-time in-depth inspection is sufficient.

Tracking the movement of bugs/tests etc. in dynamic code by correlation is the next step, allowing to ensure focus on actual bugs rather than on manually context evaluation and

correlation. One tool that has proven to be suitable in a number of large projects, e.g., the Linux kernel, the wine OS API, the openssl security library or the VLC media player, is herodotos.

5.1.3.3 Change management

Humans are notoriously bad at repetitive and monotone work like updating APIs in a large code base. This will happen, and the later in the development it happens the more likely it is that in the overall complexity of the code base and the "last minute panic" subtle semantic changes occur that raise security critical corener cases to the level of exploitable bugs.

As described above the formal specification can help ensure that the code at the specific location is in sync with the specification, but how to keep the API semantically (not just syntactically!) in sync over a large code tree? Tools that allow these must not only understand complex semantics of code, but also allow to detect isomorphic code constructs and nested constructs. One such tool that has been in wide use in the Linux kernel is coccinelle. This semantic patch tool allows specifying a semantic change and the contextual specification and then automatically generate a patch that can then be applied to alarge code based with the automatic tool spatch.

As an example here is the update of a Linux kernel api:

```
// Copyright: (C) 2009 Gilles Muller, Julia Lawall, INRIA,
DIKU.  GPLv2.

@has_sc1@
@@

#include <linux/serial_core.h>

@has_sc2@
@@

#include <linux/serial_8250.h>

@depends on has_sc1 || has_sc2@
@@

- SERIAL_IO_MEM
+ UPIO_MEM
```

This seemingly trivial update of a macro is context sensitive, and this context can be suitably described by coccinelle to allow patching the right sources that satisfy the respective constraints. A further example is maybe a bit more intuitive showing an actual API change:

```
// Copyright: (C) 2009 Gilles Muller, Julia Lawall, INRIA,
DIKU.  GPLv2.

@@
struct device dev;
expression E;
```

```
type T;
@@

- dev.driver_data = (T)E
+ dev_set_drvdata(&dev, E)

@@
struct device *dev;
expression E;
type T;
@@

- dev->driver_data = (T)E
+ dev_set_drvdata(dev, E)

@@
struct device dev;
type T;
@@

- (T)dev.driver_data
+ dev_get_drvdata(&dev)

@@
struct device *dev;
type T;
@@

- (T)dev->driver_data
+ dev_get_drvdata(dev)
```

Thus all semantically legal case of the old call dev->driver_data are properly descried and handled - thus ensuring that all instances of the old API are handled (call by value, call by reference and casted versions)

It should be noted that this is not a lexical description but actually a semantical description, thus this will match in all representations that are legal in the C language used in the Linux kernel.

5.2 Verification of Isolation Properties

Isolation properties are the main and basic properties of the virtualisation layer. These properties can be guaranteed under the assumption that the underlying hardware is trusted. It means that the internal processor registers will work properly as well as the clock and timers and other low level mechanisms. Assuming this correct behaviour, the virtualisation layer has to extend it to the upper levels (partitions).

For the temporal and spatial isolation purposes, it is assumed:

- The access to the processor registers is only allowed when the processor is in privileged mode. The processor mode can set/unset by accessing the control processor status (PMS).
- The memory control using the MMU will raise an exception when a instruction tries to write in protected areas. A memory area has associated a set of permissions that allow to control who can read/write the memory regions.
- A specific timer is used by XtratuM to control the slot duration.
- I/O accesses are controlled building the appropriated I/O maps for each partition jointly with its rights as defined in the configuration file. An exception could be raised when a partition tries to access to non allowed I/O ports.
- The interrupt vector is handled exclusively by XtratuM. Its access/modification can be done only when the processor is in privileged mode.
- XtratuM is executed in privileged processor mode whereas partitions are executed in user processor mode.

On the other hand, the configuration vector (system configuration) specifies in XML the resources allocation which contains the five main elements:.

- Hardware. Specifies the board resources: CPU frequency, memory available, type of memory, devices, etc.
- Hypervisor. Specifies the list of memory regions allocated to XtratuM, health monitor actions to be done when exceptions are captured and if they are logged or not.
- Partition Table. Specifies the partition elements:
    - Partition flags: specify if a partition will be booted by XtratuM, is a supervisor partition, uses the floating point unit, etc.
    - Amount of CPU allocated
    - List of memory regions and access rights
    - List of IO Ports allocated and access rights
    - List of ports to perform inter-partition communication
    - List of hardware interrupts allocated
    - List of devices handled by the partition
    - Trace size and allocation
- Scheduling Plan. Specifies the plan to be executed. It can include several modes and for each mode the basic scheduling policy is a cyclic scheduler. For spare time other policies can be specified.
- Channels A list of channels which define the port connections. For each channel, the following information is specified: channel identifier, type, input and output ports, maximum message size, maximum number of messages (queuing channels).

The configuration vector is seen as a contract between the system designer and the platform. This configuration vector is compiled for a specific system deployment and

attached to the hypervisor code. The data structure obtained is seen by XtratuM as the information source to guarantee the temporal and spatial isolation of the partitions.

5.3 Spatial isolation

Spatial isolation implies:

- Partitions cannot access to other memory addresses different to the allocated in the configuration vector with the rights defined
- Partitions cannot access to other IO Ports different to the allocated in the configuration vector with the rights defined
- Partition cannot use ports that are not defined in the configuration vector and its use is coherent with it (source or destination)
- Partitions cannot handled interrupts different that the allocated in the configuration vector
- Partitions cannot use devices different that the allocated in the configuration vector.

XtratuM will enforce all the hardware mechanism to guarantee that the spatial isolation is guaranteed.

5.4 Temporal isolation

Temporal isolation implies:

- Partitions are executed under a cyclic scheduler. The cyclic scheduler specifies a schedule plan which consists in
    - A Period: it is known as Major Frame (MAF) and is defined by the system designer. Usually, the MAF is the lowest multiple period of the periodic activities.
    - Temporal window: specified as an offset with respect to the MAF and duration.
    - Partition: partition to be executed
- The schedule plan can contain not used temporal windows. This is assumed as spare time not assigned to any partition.
- Partitions are executed only in the temporal windows (slots) specified in the configuration vector.
- A partition will not be able to use the system resources if the clock does not match with its temporal windows.
- If other policies are specified for using the spare time, those partitions that have specified this policy in the configuration vector will be candidate to be executed during the duration of the spare time.
- Other policies are limited to **round robin** between all partitions interested in the spare time, or **priority based**.

5.5 Properties verification

The configuration vector included in the deployment specifies the behaviour of the system. So, the hypervisor state is defined by:

- The current partition identifier
- The absolute time of current MAF origin
- The current clock value
- The current slot
- The current scheduling mode
- The hardware mechanisms: memory areas, I/O maps, interrupt vector, etc.
- Additional variables not directly related to the properties

The hypervisor can be invoked as consequence of a hypercall (a partition requests a service) or an external interrupt or a trap occurs or temporal window has reached its end. In all these cases except the temporal window end, the conditions associated to the execution remains. It means that the same partition will still be executed at the end of the service or interrupt.

In the case of the end of a temporal window, the hypervisor has to switch to execute another partition. In this case, the new set of variables associated to the state has to be read from the configuration vector and maintained during the next temporal window.

In order to validate the temporal and spatial properties, the hypervisor has to check that its state is coherent with the configuration vector considering the current time. It means that given a clock value and an execution mode, it determines:

- The current slot that should be under execution: comparing the current slot with the specified in the configuration vector
- The current partition that should be under execution: comparing the current partition with the specified in the configuration vector
- The memory areas and IO maps: comparing the current values with the specified in the configuration vector
- The interrupt vector: comparing the current value with the specified in the configuration vector

These comparisons determine the **pre-conditions** to be evaluated each time the hypervisor is invoked. Once these conditions are successfully evaluate, the hypervisor executes modifies these variables (i.e. to access to all memory maps), and executed the internal service.

These comparison should be evaluated at the end of the hypervisor execution (**post-conditions**) in order to guarantee that the partition is executed again (or the new partition in the case of a new slot) is going to be executed with the values specified in the configuration file.

Some other isolation mechanisms that are related to offered services as hypercalls: The parameters are analysed by the hypercall service in order to guarantee the isolation. In this

situation, the memory copy (XM_memory_copy) hypercall permits to copy a memory block from a memory region to another. In this case the hypercall validates the parameters against the configuration vector to allow the copy or reject the service.

6 Conclusion

This document presented the methodologies and their dependant technical requirements to validate an OVERSEE platform implementation.

Three main categories of validation have been described. The Conformance Testing aims to validate communicating systems and their interfaces with respect to their specification. Common Criteria defines a way to ensure that some security properties are enforced by an implementation. Model Checking and verification techniques provides guarantee that a model and its implementation enforces some critical properties as spatial and temporal isolation.

This work will serve as an input for WP4 in which will be decided what has to be tested, the used methodologies and the tool selected. Moreover, as soon the decision taken, WP4 will prepare the support of validation by the implementation in cooperation with WP3.

7 References

[1] European Telecommunications Standards Institute (ETSI). (1995) ETS 300 406: Protocole and Profile Conformance Testing Specifications - Standardization Methodology. [Online]. http://portal.etsi.org/mbs/Referenced%20Documents/ets_300_406.pdf

[2] European Telecommunications Standards Institute (ETSI). (1995) ETR 266 standard: Methods for Testing Specification (MTS) - Test Purpose Style Guide. [Online]. http://portal.etsi.org/mbs/Referenced%20Documents/etr_266.pdf

[3] International Organization for Standardization/ International Electrotechnical Commission (OSI/IEC). (1995) ISO/IEC 9646 standard: Information Technology - Open Systems Interconnection - Conformance Testing Methodology and Framework. [Online]. http://webstore.iec.ch/servlet/GetPreview?id=39516&path=info_isoiec9646-7en.pdf

[4] European Telecommunications Standards Institute (ETSI). TTCN-3: Testing and Test Control Notation. [Online]. http://www.ttcn-3.org/home.htm

[5] Common Criteria v3.1. Release 3, Part 1: Introduction and general model. [Online]. http://www.commoncriteriaportal.org/files/ccfiles/CCPART1V3.1R3.pdf

[6] Common Criteria v3.1. Release 3, Part 2: Security functional requirements. [Online]. http://www.commoncriteriaportal.org/files/ccfiles/CCPART2V3.1R3.pdf

[7] Common Criteria v3.1. Release 3, Part 3: Security assurance requirements. [Online]. http://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R3.pdf

[8] Common Criteria v3.1. Release 3, Evaluation Methodology. [Online]. http://www.commoncriteriaportal.org/files/ccfiles/CEMV3.1R3.pdf

[9] (2011, April) Spin HomePage. [Online]. http://spinroot.com/spin/whatispin.html

A. Selection of Modules - Methodology Notes

Open Source is fundamentally different than traditional commercial software in that there generally are multiple versions of (constructively) competing software offerings that can satisfy a specification. Due to the nature of pre-existing software though we have a different selection process to look at:

- identify potential candidates
- Evaluate the suitability/stability/roadmap, etc.
- consider integration issues (technical and non-technical i.e. license)

Obviously a simple ad-hoc selection of open-source components will not due in a complex hardware software system, thus a systematic evaluation is needed. Notably as OVERSEE targets a system with reliable system level security properties swell as the ability to achieve later safety certification (even if currently de-scoped due to effort limitations) the selection needs to be based on criteria not only covering functional but also non-functional requirements.

With RH Enterprise having achieved EAL Level 4 certification (with a very specific configuration - utilizing SELinux as the core LSM along with a number of other facilities at the VFS and network level) it seems reasonable to argue the reuse of "proven-in-use" components in the context of and entity anticipating an OVERSEE-platform certification. To support such efforts our selection of components must be arguably sound and documented.

A.1.1 Assess the complexity of the underlying mechanism and its adaptability

This was done by first analyzing the respective sources (and condensing this into a presentation set), next a minimal prototype LSM (OVERSEE_lsm) was written that basically added a trivial security hook to the relevant inter-partition communication extension provided by XtratuM and the respective paravirtualization layer in the Linux kernel. This hook was integrated and tested more or less stand-alone - functionally it was reduced, limiting the action of the LSM to a pure reporting interface:

<snip> kernel dmesg output (TODO: cut&past)

A.1.2 Assessment of impact of modifications

As the code changes are minuscule and the interface to the LSM is generic (that is more or less the same for all Linux LSMs currently supported), further the changes are well encapsulated in the paravirtualization extension of the Linux kernel - provided by the appropriate XtratuM kernel patch - the impact can be analyzed by local inspection. Note that while these changes might lead to performance or stability impact, they are not expected to break the security mechanism logic which is the key point with respect to security. This can be assumed as the interface is well specified and thus provides a "contract based" model of exchanging an existing LSM by an extended LSM.

```
LoC needed for minimal LSM: < 250 LoC
Number of files changed: 7 (includeing Makefile/Kconfig)

     if enabled:
       security_* ----> include/linux/security.h
         |
         `-> security/security.c
                |
                `-> security/trivial/trivial_lsm.c
```

(note trivial/trivial_lsm.c is the prototype OVERSEE lsm implemented for demo purposes and presented at the bochum meeting)

A.1.3 Assessment of available modules

There are a number of LSM modules integrated in the Linux kernel, with varying capabilities and notably substantially varying complexity and thus runtime impact. The available in-tree LSM are

Goals of LSM

Security is a main concern in GPOS - naturally GNU/Linux has been focused on security issues - after all that is one of the things that distinguishes it from its main competitor... These efforts have been focused on the kernel level in the 2.4 kernel series with increasing efforts to extend it to user-space in a more formal manner in the 2.6 series of kernels - thus in early 2003 the proposal for the Linux Security Module (LSM) was integrated in the mainline Linux development and accordingly tools (kernel, user-space and configuration) developed.

- Part of mainline Linux 2.6 since December 2003 - proven-in-use?
- security framework for mandatory access control
- establish common models of MAC implementation
- minimize changes to the Linux kernel (notably prevent duplication)
- ensuring completeness of coverage by hooks (see Using CQUAL for Static Analysis of Authorization Hook Placement)

For details see http://www.usenix.org/event/sec02/zhang.html - we believe that this is a suitable basis for building a strong and robust security module (MILS architecture) on top of the XtratuM separation kernel that provides the "first-line-of-defence" in the oversee architecture. The GNU/Linux runtime provided in the secure-I/O partition (providing overall system device I/O) is thus already in a constraint environment. The final application partitions can only communicate via the secure-I/O partition thus re-enforcement of the security properties based on well established technologies including

- Linux Security Modules
- Fast User Space File system capabilities
- IP-tables / Traffic shaping

It seems a reasonable strategy to balance security demands and ease of use.

Available LSM and very brief feature list:

To summarize the evaluation that has been done in the context of the investigation efforts of WP2 and WP3, a brief listing of available security modules and there capabilities is:

- SELinux, label-based approach, integrated in Mailine Linux in early 2.6.X,
- AppArmor, path-based access control, integrated in Mainline as of 2.6.36
- http://en.opensuse.org/SDB:AppArmor, latest is 2.3
- Linux Intrusion Detection System, LIDS, www.lids.org, latest is 2.2.3rc9 for 2.6.31/32
- FireFlier, label socket with an application context, conceptually allowing the interactive creation of rules, latest (obsolete?)
- CIPSO, IETF Commercial IP Security Option, IETF CIPSO Working Group,
    - security attributes to outgoing network packets generated from applications
    - read security attributes from incoming network packets
- SMACK, path based access control, Mainline as of
- TOMOYO, path-based access control, Mainline as of 2.6.30, latest version in mainline 2.2.0 (reduced feature set), 1.7.1 (extended feature set - patch required), http://tomoyo.sourceforge.jp

Selecting the appropriate based to include our OVERSEE specific extensions is not primarily a matter of writing code but a amateur of developing suitable specific requirements and selecting the variant that has the best fit to the functional and assurance requirements. While the functional requirements assessment at the technical level should be quite clear the non-functional assessment does mandate some notes.

To develop a system that anticipates future certification it is essential to ensure that suitable evidence of functionality, stability and reliability is available with adequate confidence - CC nicely splits this in SFR and SAR - adopting this split we here focus on the SAR that can be achieved for a FLOSS component.

A.1.4 QA – Quality Assurance

- Repository traceable?
- Bug tracking?
- Mainline distribution usage?
- Mainline kernel integration?
- If not in mainline - how invasive is the patch?
- Test-suits available?

A.1.5 Technical documentation available

Allowing to expect modification/extensions to be doable with reasonable probability of not introducing systematic faults in the security related logic of the LSM.

A.1.6 Maintainability

- Roadmap available?
- Maintainer known/number of core developers?
- Community active?
- Support by vendor/commercial entity given?

Notably the maintainability criteria might seem a bit strange but it is not uncommon in FLOSS developments to have technical superior solutions that are not well supported or lack community endorsement and thus are not long-term stable. As the ways to ride dead horses is well documented it is not necessary to add any further variants of this discipline in the context of OVERSEE.

With this data at hand one then can do the final step of extending the open-source component to fit the technical needs of the OVERSEE platform and integrate it into the runtime environment. If the selection criteria have been well established and the necessary evidence based considered during selection then this can significantly aid certification (both security and safety).

B. Conformance Testing: how implementation meets specifications?

Safety References related to the usage of pre-existing software - specifically NOT developed for safety related usage.

B.1. Functional Safety related Standards

- IEC 61508-3 (edition 1998)
  - 7.4.2.7 - demonstration of independence of software with mixed SIL levels
  - 7.4.2.11 - requirements on pre-existing SW
- IEC 61508-3 (Edition 2011)
  - 7.4.2.7/7.4.2.8
  - 7.4.2.11/12 - requirements on pre-existing SW
- IEC 61508-7 (edition 1998)
  - B5.2 Black Box testing (XM test suite)
  - B5.4 Field Experience
  - C5.1 Probabilistic Testing

IEC 61508 is a guiding, generic functional safety standard. Components certified to IEC 61508 can be re-used in the context of derived standards i.e. IEC 26262 (naturally with limitations pertaining to the specific context). Thus taking, the generally far more strict, regime of IEC 61508 as a guidance for the argument development for COTS/FLOSS components allows a reasonable probability of later certification efforts to be successful with tolerable effort. Though OVERSEE is a prototype implementation of a technical solution and not a to-be-certified product.

- IEC 26262-6:
  - 8.4.5 swell as table 10 and 11
  - 7.4.6 categorization of COTS for safety (EN 50128 3.18 open-source = COTS)
  - 7.4.8 -> COTS certification 26262-8 Clause 12
- IEC 26262-8:
  - 14 Proven in use argument explicitly noting:
    - Candidate being used in other safety-related industries; or
    - Candidate being a widely spread COTS product not necessarily intended for automotive applications.
  - All of clause 14 can and shall be applied.
- IEC 26262-9
  - 5.4.6 ASIL decomposition

Note that it is not an intent of OVERSEE to provide certification rather it is the intent of the OVERSEE development to, with reasonable probability; prevent show stoppers from being introduced that make a certification impossible. The clauses mentioned here are not being followed in any formal way rather they have been reviewed to assess the feasibility of certification of key OVERSEE components in the context of a safety assessment building on pre-existing (COTS) arguments. Note further that EN 50128 (prEN 50128 2009) explicitly identifies open-source as being equivalent to COTS in the context of safety related systems (Clause 3.18) though EN 50128 is a rail standard, it is, being a transport standard, suitable for cross-referencing when arguing suitability (might add that the safety requirements in Rail are fundamentally more restrictive than the watered-down safety requirements for automotive industry...)

B.2. DO 178B Software Considerations in Airborne Systems

- Section 2
- Subsection 2.4 (COTS issues)
- Section 3
- Subsection 3.2 (specifically component-Z example)
- Section 12
- All of clause 12 is of relevance

Obviously DO 178B is not applicable to OVERSEE directly though due to this standard being well accepted and concise (notably compared to the mess that IEC 26262 provides) it is a good starting point for introducing safety considerations in a project heavily building on COTS/FLOSS components. Further due to XtratuM being targeting the "DO 178 market" and the design being built on ARINC 653 (a prime Avionics standard for partitioning systems) it seems natural to consider DO 178B as far as reasonably possible.

B.2.1 **ARINC 653**

Without further details - XtratuM the separation kernel in use in OVERSEE was designed and implemented along the guidelines of ARINC 653. Being suitable for the avionics domain up to Level A certified systems/components it seems more than suitable for the automotive domain (a subjective claim would be that ASIL D is at best DO 178 B/C - see clause 2.2.1/2.2.2).

B.3. IEC 15408 Common Criteria

A common criteria certification is against a specific protection profiles (PP) and in this sense one cannot globally state that Linux has been certified to any EAL level. But taking the current certification reports, which show that Linux has been certified to EAL4+ against the following number of PPs - specifically:

- General Purpose Operating System Protection Profile (GP-OSPP)
- Controlled Access Protection Profile (CAPP)
- Labeled Security Protection Profile (LSPP)
- Role-Based Access Control Protection Profile (RBACPP)

it is legitimate to claim that GNU/Linux can be certified to EAL4+.

Due to the role of security in OVERSEE we briefly outline The overall technologies for certification of GNU/Linux against specific PPs that are in use (and available to the GNU/Linux based partitions of OVERSEE):

System wide governing Security Policy providing:

Domain/Type enforcements, MILS. This is provided by various LSMs currently available (SELinux, SMAK, TOMOYO, etc). As OVERSEE is based on a separation kernel as the primary isolation enforcement (spatial and temporal) this constitutes the second level of security mechanisms. To effectively make use of the core mechanisms for security policy enforcement at the partition level notably the secure I/O partition which is based on GNU/Linux, extensions to the existing LSM (Linux Security Modules) is needed to account for the kernel level extensions provided by the paravirtualization and the inter-partition communication mechanisms. Rather than extending these to generic interfaces (i.e. sokets) the extension of the LSM to intercept sensitive processing steps is being implemented (i.e. at the queue management level of the virtual device layer in Xtratum)

Access Control covering the primary user accessible resource in UNIX - files:

The layering of control policy enforcement in GNU/Linux is roughly traceable to the historic development:

- Credentials
- Capabilities
- Namespaces
- Attributes

Namespaces are provided for multiple facilities at kernel level, and can also be enforced by VFS extensions (i.e. FUSE), attributes are a well known mechanism, notably extended attributes provided in GNU/Linux as xattr and acsl in the context of Mandatory Access Control scemes (i.e. SELinux). The capabilities cover the full spectrum of DAC, MAC and RDAC (Role Based Access Control).

Memory Protection:

Memory protection comes in two levels in OVERSEE, at the core level XtratuM provides a fundamental separation of memory areas at the partition level thus ensuring that security properties at a pro-partition level actually can be established. A second "line of defense" is then the usual OS level memory protection mechanisms - with potentially varying capabilities and maturity.

GNU/Linux has a plethora of technologies fielded which address threats related to memory protection issues, a few of which are PaX, ASLR (in mainline) ProPolice and Stack-smash-protection as well as chroot jail hardening methods.

These methods allow to reinforce the GNU/Linux partitions of OVERSEE, notably the secure I/O partition as this is obviously a major potential threat area.

Identification and Authentication:

Identification and Authentication is again covered by a broad range of facilities tightly integrated in all mainline GNU/Linux distributions, notably PAM (pluggable authentication modules) which is managed at kernel.org, appropriate account management as well as authenticated user access.

In OVESEE the first level of identification is provided by XtratuM, as ARINC 653 mandates, xtratum provides a static port attributes provided by the XtratuM core. These port attributes include a unique partition identifier and a unique port name thus providing the first level of identification in interpartition communication. Authentication is then at the partition level (provided by services within the partition), building on secured data provided by the security service partition.

Cryptographic services:

Any sound security system needs to provide core services based on a certified random number generator. These core services are provided by the security service module and, where necessary, based on the hardware security module.

services: Cipher-based MAC (CMAC), Hash-based Message Authentication Code (HMAC), Signature Verification/Generation, Cipher