D4.1

Security testing framework: strategy and approach

Project number:	731456
Project acronym:	certMILS
Project title:	Compositional security certification for medium to high-assurance COTS-based systems in environments with emerging threats
Start date of the project:	1st January, 2017
Duration:	48 months
Programme:	H2020-DS-LEIT-2016

Deliverable type:	Report
Deliverable reference number:	DS-01-731456 / D4.1/ 1.0
Work package contributing to the deliverable:	WP 4
Due date:	Sep 2017 – M09
Actual submission date:	29th September, 2017

Responsible organisation:	UROS
Editor:	Thorsten Schulz
Dissemination level:	PU
Revision:	1.0

Abstract:	Approach, strategy, and architecture for the implementation of security testing framework are proposed.
Keywords:	Security framework, security testing, analysis, fuzz-test methodology

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731456.

Editor

Thorsten Schulz (UROS)

Contributors (ordered according to beneficiary numbers)

Andreas Hohenegger, Staffan Persson (atsec)

Alvaro Ortega (E&E)

Reinhard Hametner, Michael Paulitsch (THA)

Caspar Gries, Sergey Tverdyshev, Holger Blasum (SYSGO)

Tomáš Kertis (UCO)

Thorsten Schulz (UROS)

Executive summary

This deliverable evaluates the state of the art in security testing techniques in relation to relevant industrial security standards. The research starts with a survey of non-industrial security frameworks and general identification of security vulnerabilities. We then have a look into the different testing contexts covered by Common Criteria requirements and IEC 62443 standards and certification schemes. This is also brought into context with the certMILS application pilots and the applicability to testing of compositions as the fundamental architecture of a MILS system.

Future work will focus on refining features of the testing framework for security testing of operating system components. The strategy for these activities is outlined in the third chapter, together with a short study of the technical feasibility.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Security describes the resistance to threats, intentional and unintentional. Threats always exist and target all systems.

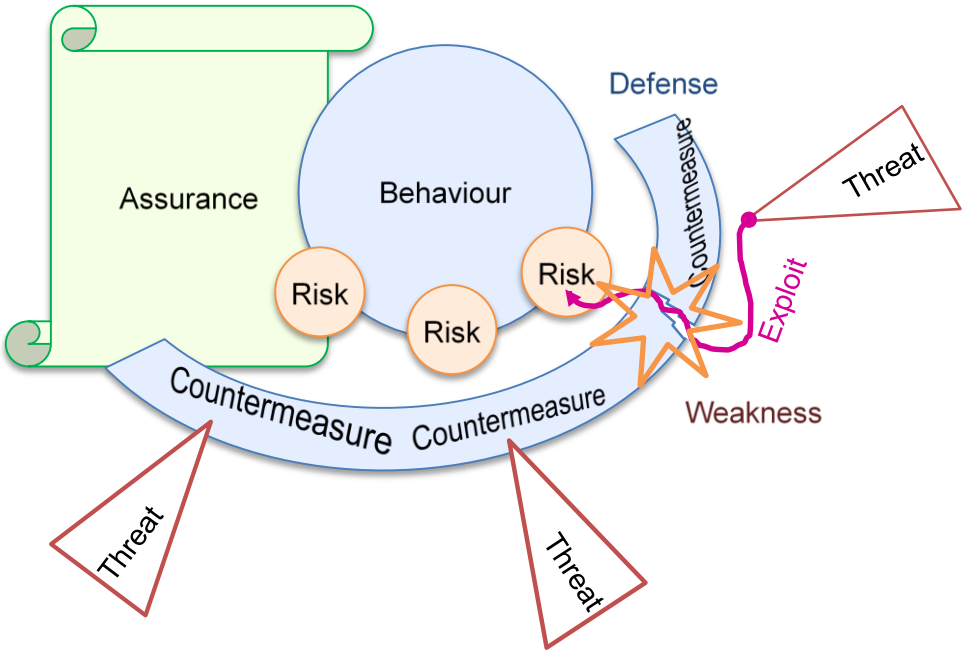

Figure 1: The interplay of the concepts composing the security realm.

Assurance is a measure of confidence in the correct behaviour of a system. A vulnerability is a weakness of a system, which can be used to cause a threat. A threat is a method leading to a dangerous event, i.e. risk. The defence is the collection of countermeasures preventing a threat escalating to a risk. An exploit bypasses the defence, using a vulnerability to realize a threat leading to a dangerous event. It results in loss of integrity of the system and its assurance.

Security frameworks have gained large attention in web applications. Often, software tests for “assurance-less” products are not as rigorous and possibly not required for functional acceptance. The applications too often expose a large attack surface to the whole internet. Nonetheless, the operational environment changes very rapidly, making maintenance over the full software lifecycle very costly, if not impossible, for individual solutions.

Larger companies and interest groups have thus developed security frameworks for their target market to improve overall reliability of products generated from their bases. Examples are: OWASP Testing Framework[1], Microsoft Security Development Lifecycle (MS-SDL), SAGE ("Scalable Automated Guided Execution“)[2] and SLAM[1]. In addition as a different concept, Google has established the OSS-Fuzz environment[3] to donate their server resources to security test open source software.

[1] SLAM is a project for checking that software satisfies critical behavioral properties of the interfaces it uses. Static Driver Verifier is a tool in the Windows Driver Development Kit that uses the SLAM verification engine.

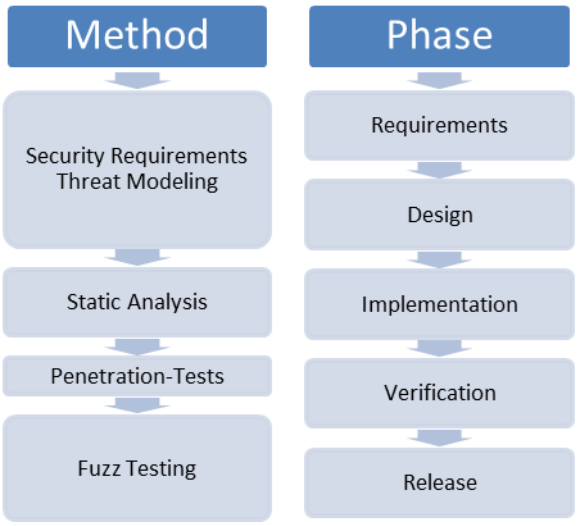

Figure 2: Mapping of security test method to development phase (MS-SDL and [1]).

According to Pohl[4] in Figure 2, security frameworks should consist of multiple tools accompanying the whole development lifecycle: starting with Threat-Modelling in the early requirements and design-phase, static analysis tools (see certMILS deliverable D1.2, in preparation) in the implementation phase and dynamic analysis tools (e.g. Fuzzer) in the verification phase.

The ISA standard for “Security in industrial automation and control systems” IEC 62443[5] defines similar concepts in the subset 1-1 for general “concepts and models” compared to Figure 1. The security context is set as the two connected processes of assurance and assessment, as shown in IEC 62443, Figure 3.

1.1 Security testing techniques in publications

Security testing uses many techniques, which are detailed in the next chapter. One of the currently most researched techniques is fuzz-testing. Oehlert [6] names fuzz-testing a technique, “to better ensure the absence of exploitable vulnerabilities” by “checking large numbers of boundary cases”, that functional testing cannot cover. It adds *negative test cases* to verify, that a software or “product does not do something it shouldn’t do”. This aspect is also detailed in [7] and displayed in Figure 3.

In other words, this still requires classic functional testing to provide verification with full coverage. However, to leverage comparable negative test cases, it is imperative for effective fuzz-testing to get feedback from code coverage analysis to become a quantifiable technique.

Fuzz-testing is a computational and thus time-consuming technique. Even with code coverage information it is hard to generate test cases, that dive deep to achieve good coverage. The combination with other techniques is proposed in [8]: “Static Exploration of Taint-Style Vulnerabilities Found by Fuzzing”. This work locates initial weaknesses through fuzzing with known fuzz-testing tool AFL[9] and tools provided by the LLVM compiler toolchain[10]. Found weaknesses are converted to vulnerability patterns and matched against the complete source base with static analysis. The research found asserted a high rate of false positives as a drawback, which the authors mitigated by a ranking algorithm and also identified this for future improvement, e.g. by using symbolic execution and path reachability diagnostics.

In [11] different tools are enumerated for concurrency and multithreaded testing. For random and fuzzing is a tool “contest”, “forcing threads to interleave at random intervals. As fuzzing strategies encounter a faulty set of thread interleaving by chance these might increase the chances of finding a bug. However they provide no guarantee regarding the detection of race conditions.”

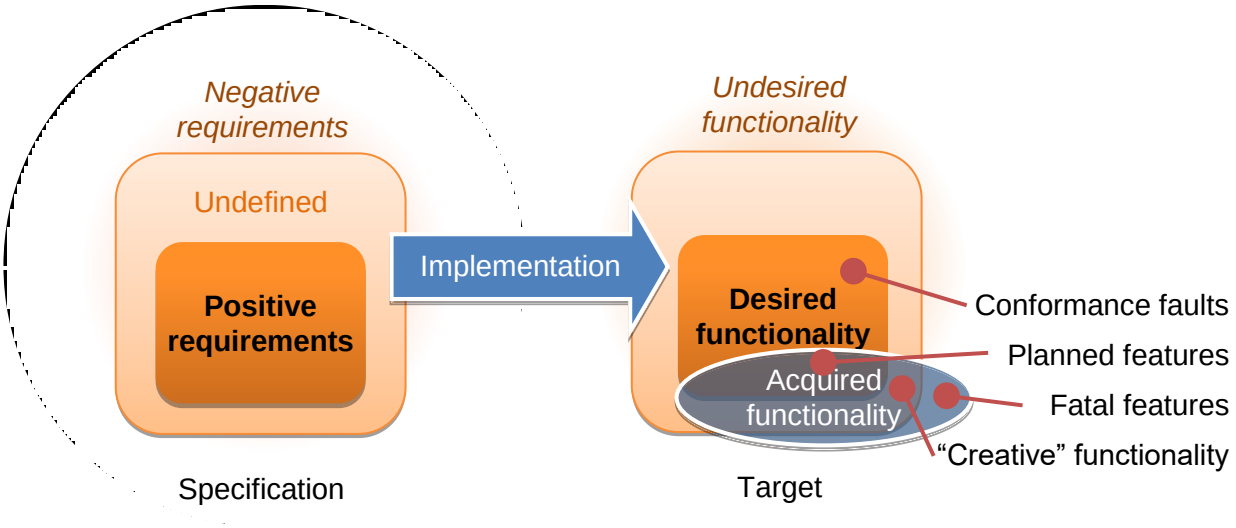

Figure 3: The justification for and target of negative testing (redrawn from [33]).

Microsoft holds a patent [12] claiming "Amplification of dynamic checks through concurrency fuzzing". An application is prepared with additional dynamic checks to monitor runtime behaviour under the influence of randomization of thread scheduling. A related tool is Microsoft's "Cuzz" which is part of the AppVerifier Suite. "Cuzz is a very effective tool for finding concurrency bugs. Cuzz works on unmodified executables and is designed for maximizing concurrency coverage for your existing (unmodified) tests. It randomizes the thread schedules in a systematic and disciplined way, using an algorithm that provides probabilistic coverage guarantees." Licensing and availability of Microsoft-tools make them inappropriate for use in certMILS. Though similar techniques are already in development as part of the functional testing of the certified hypervisor operating system PikeOS, as used in the context of the certMILS project.

Takanen et al. [7] also go into detail about fuzzing of industrial automation systems. The best known and preferred tool at the time of writing in 2008 was "Achilles", which is also described in deliverables D1.1[13] and D1.2 (in preparation). Correlated, the author divides SCADA[2] fuzzing into control functionality blocks of Ethernet communication processing, logic processing, and I/O processing. They consider how different stress types in hostile (networking) conditions and environments affects devices. A SCADA device must maintain its safety policy, e.g. hostile stress should: have no affect at all, trigger fail-safe mode, etc. Challenges in fuzzing SCADA systems as noticed by the authors:

- protocol diversity
- implementation ambiguity
- [special] equipment access
- configuration complexity
- test simulations either with and without load
- grey-box access to SUT
- multi-way redundancy
- fail-over behaviour of SUT
- performance constraints of SUT
- accounting for watch-dogs, fail-safe modes, communication failover

[2] Supervisory Control and Data Acquisition (SCADA) is a subset of IACS, but often used to refer to IACS as well.

These challenges will influence solutions for the testing framework, especially concerning the scope that can be tested reproducibly. Many of the above points from the cited publication result from looking at products and systems and have a lesser impact on individual components, such as those of a MILS system.

1.2 Purpose of this document

"Describe the approach, strategy, and architecture for the implementation of the security-testing framework."

The Security framework must raise the assurance of a component, throughout and especially at the beginning of the lifecycle of the system. By providing an additional security-testing infrastructure, the integrator of a MILS system can demonstrate the required properties of the component in their product/solution to

- Fulfil other component's requirements regarding the environment or non-interference properties other components cannot fulfil themselves.
- Gain assurance of interface robustness.
- Maintain assurance level in case of a component update.

Chapter 2 Testing contexts

This section discusses existing contexts, and their objectives and methods for achieving assurance.

2.1 Common Criteria testing requirements

The Common Criteria for Information Technology Security Evaluation (CC) [14], or ISO/IEC 15408, represent a flexible framework for the evaluation of the security of IT-products, or a part of it that is called the target of evaluation (ToE). The standard is flexible in that it applies to a broad range of ToEs. Furthermore, developers (or sponsors of an evaluation) may customize the detailed CC evaluation activities to suit their own assurance needs. To this end, they choose from two distinct sets of requirements. One of these consists of security functional requirements (SFRs) to model the properties of the ToE under investigation. The second one represents a collection of assurance requirements that the developer may choose to impose on the ToE (SARs). Two important classes of the latter set of CC requirements address security testing:

- Testing of the security functionality (security functional testing), is described as part of the assurance class ATE;
- Penetration testing is part of the vulnerability assessment described by the assurance class AVA.

Security functional testing provides assurance that the ToE behaves as modelled using the SFRs. This set of security functions must also be reflected by the developer provided information (in the functional specification, ToE design and implementation representation). On the other hand, penetration testing is the testing for potential vulnerabilities to determine if they are exploitable by a hypothetical attacker.

Like all SARs, testing is described in CC Part 3 [15]. Both, the ATE class and the AVA class consist of “families” which cover different aspects of each. It is important to note that, due to the generic nature of the CC, requirements of these classes do usually not appeal to specific testing methods, or even testing tools. However, independent of how the testing is performed in detail, it states – in abstract language – what needs to be tested and how the quality of the testing is to be assessed. In addition, it requires the developer of the ToE to provide dedicated documentation of his tests, and the evaluator to perform independent testing as well as the vulnerability analysis to a certain degree of rigour.

The CC allow to consistently choose the detailed SARs by selecting an evaluation assurance level (EAL1 – EAL7) for the evaluation. In this case, the EAL also determines the level of testing. With increasing EAL, more rigorous components of the different ATE families must be met in successful evaluations. They require increasing test coverage in terms of security functionality, as well as test depth. This should provide more confidence that the security functionality works as specified.

Testing must be repeatable and reproducible. This means that the tests must show the same result when repeated by the same tester, but also when performed by someone else. Therefore, tests have to be not only deterministic, but also sufficiently documented to guarantee reproducibility by other testers.

Testing must also be performed under conditions that are consistent with the evaluated ones, since the functionality may depend on the environment and change with the configuration of the ToE. If there are different versions or configurations of the ToE, all of them have to be tested unless it can be demonstrated that the security functionality of the ToE will remain unchanged.

2.1.1 Testing in the Common Criteria context

The Common Criteria divide test requirements into four areas. These are represented by the families of the ATE class:

ATE_FUN addresses requirements on the test documentation;

ATE_COV is the test coverage analysis of the security functions and their externally visible interfaces (TSFIs);[3]

ATE_DPT is the test depth analysis of the testing of ToE subsystems implementing the security functions;

ATE_IND is the independent testing performed by the evaluator (in addition to the developer testing). For this family, the developer only has to make the ToE and the test environment available for independent (evaluator) testing.

As an example, the specific requirements for EAL3 are outlined in Table 1.

Table 1: Common Criteria testing requirements for EAL3 (see also CC Part3 Table 1 [15])

Assurance family and components	Details
ATE_COV.2	The objective is to confirm that all of the TSFIs, described in the functional specification, have been tested.
ATE_DPT.1	The objective is to confirm that all TSF subsystems, described in the ToE design, have been tested. The subsystem descriptions of the TSF provide a high-level description of the internal workings of the TSF. Testing at the level of the ToE subsystems provides assurance that the TSF subsystems behave and interact as described in the ToE design and the security architecture description.
ATE_FUN.1	The objective is to confirm that the functional testing performed by the developer are performed and documented correctly. The test documentation shall consist of test plans, expected test results and actual test results. This includes instructions for using test tools and suites, description of the test environment, test conditions, test data parameters and values.
ATE_IND.2	The objective for independent evaluator testing is to confirm that the developer performed some tests of some interfaces described in the functional specification. The evaluator first performs a sub-set of the developer test and then devises, documents and executes own tests.

The functional testing, as described in ATE, only deals with testing of the correctness of the security functionality. This means that with increasing EAL we obtain an increasing level of testing of these functions and, henceforth, increasing confidence in that these functions work as intended. As an example, at EAL2 the ATE_COV.1 is selected instead of ATE_COV.2. ATE_COV.1 requires

[3] In the CC context, the portion of the ToE implementing security functions is referenced as TSF and the interfaces by which the user, or non-TSF portions of the ToE, interact with the TSF are called ToE security functional interfaces (TSFIs).

only some of the TSFIs to be tested. Even more, at EAL1 the ATE_COV family would not be included at all.

The CC also has the concept of being able to resist a certain attack potential, but there is no formal or direct relationship between the level of testing and the ability to resist an attacker with a given skillset.

2.1.2 Vulnerability analysis in the Common Criteria

The resistance to certain attack potentials is formally a part of the vulnerability analysis. This assessment determines whether an attacker could violate any of the SFRs used to formulate the security functions of the ToE. This could include bypassing, tampering monitoring or any other form of abuse. The corresponding security assurance class AVA has a single family, AVA_VAN, which describes this evaluation aspect. As part of it, the evaluator needs to conduct penetration testing. If the modelled security functionality includes e.g. the protection of a user's data from the access by others, covert channel analysis should be considered in the assessment. Again, the CC are unspecific when it comes to the methods employed to conduct the pen-tests.

Note that penetration testing in the CC is not to be confused with black box testing. In the context of AVA_VAN, all sources of input from the evaluation are used. With increasing EAL, more information is available for the analysis and more rigour is applied in its assessment. The analysis varies from a survey of public domain information, to independent analysis, to independent and focused, and finally to independent, methodical.

Any identified potential vulnerability is then measured, as to whether it can be exploited and, if yes, what attack potential is necessary to do so. The main issue with vulnerability assessments is not the measurement of exploitable vulnerabilities, but to identify them in the first place. Especially, at lower assurance level with limited design documentation and no access to source code, it is hard to find vulnerabilities. On the other hand, when such information is available at higher assurance levels, it is more likely that potential and exploitable vulnerabilities can be identified.

The CC define the following attack potential levels associated with the different EALs:

- EAL1 – EAL3 **basic** attack potential.
- EAL4 – **enhanced-basic** attack potential.
- EAL5 – **moderate** attack potential.
- EAL6 – EAL7 **high** attack potential.

The respective attack potential is associated with a certain combination of expertise, motivation and resources of the attacker. An empirical method to compute it is offered by an appendix of the Common Methodology for Information Technology Security Evaluation [16], which accompanies the CC. The evaluator assumes the computed potential when conducting the penetration testing. Furthermore, it is taken into account when the assurance components for the evaluation are chosen, e.g. by means of an EAL. Thereby, the EAL for the entire evaluation may be related to the assumed attack potential.

2.2 IEC 62443 testing requirements

The set of standards IEC 62443 [5] addresses issues of security for IACS and is derived from the SDLA, which is in turn based on established standards such as CC, OWASP, CLASP, IEC 61508 (functional safety) and DO-178B. The scope of the standards defines specific roles of the asset owner, the system integrator and the product supplier. The latter develops and designs the product for the intended environment. The integrator then integrates a configured component into its automation solution that the asset owner operates and maintains. The standard defines duties concerning the product security life cycle for each role. These are covered by the different sub-standards of IEC 62443.

Much of the testing for security is covered in the component development by the product supplier and the system integrator. Though testing is also later part of the patch management (see part 2-3) involving all parties. Patching of an operational, potentially suspended for maintenance, solution may be assisted by a testing framework to help and complete the complex procedures. Tasks of the update management are already rooted at the component level in part 4-1 (see Table 2). Here in PM-1, testing is an important measure to assure continued defence in depth and absence of compromising side-effects.

Table 2: Requirements concerning update management throughout the product life cycle (IEC 62443-4-1:12)

Name	Description
PM-1: Update qualification	demonstrate correctness, absence of regressions, involving the developer and all suppliers
PM-2: Update documentation	detailed update documentation
PM-3: Dependent component update documentation	document compatibility to other component's updates
PM-4: Update delivery	guarantee authenticity of updates
PM-5: Timely delivery of patches	update deployment infrastructure must not contain impediments

Asset owners may also be confronted with tests for security functionality verification (CR-3.3, -4-2). These tests verify the operation of security controls and may run application during normal operations. Asset owners, i.e. operators, have to be aware of these tests to understand their implications, e.g. triggering an intrusion detection system or checking the function of audit logs. CR-3.3 notes, that the design should not affect safety functions, which must be considered in the overall security architecture.

The following sections cover the security testing requirements on component testing and testing of composed systems that arise from applying IEC 62443-3-* (system level) and IEC 62443-4-* (component level), e.g. to achieve EDSA ISASecure scheme certification.

2.2.1 Railway pilot testing requirements

The railway demonstrator for the certMILS project is the TAS Platform 2.x included in the Thales product "CyberGate". Future work of the project certMILS will apply the security framework to this platform.

The tests run by the testing framework shall demonstrate that the defined secure reference configuration for the system is applied. This reference configuration is defined during the design phase of the development process. The tests shall be done against the requirements based on IEC 62443-4-2 (e.g. user authentication). Furthermore, newly listed CVEs (Common Vulnerabilities and Exposures) related to the ToE shall be identified and checked. Also already fixed CVEs shall be checked against recurrence and absence in the system. CVE is a standardized naming convention for security vulnerabilities in IT systems including all major vendors. The list of CVEs [17] is freely accessible.

Different methods are defined in the IEC 62443 standard and shall be applied to the railway pilot:

- Security requirements testing, vulnerability testing and penetration testing.

- For example, white box security tests for testing secure configuration. Black box tests for testing interfaces such as ports outside of the ToE or the complete system under consideration.
- Review methods for showing the application of processes based on IEC 62443-4-1.

See the definition of the testing methods in IEC 62443-4-1, also included here in Table 3.

Table 3: Testing methods defined in IEC 62443-4-1, table 10.1.

Name	Description
SV-1: Security requirements testing	This testing focuses on verifying all the security requirements in the security requirements specification (SecRS) have been met. Functional, negative, boundary, performance and other types of standard testing will be performed on the security capabilities in the SecRS.
SV-2: Threat mitigation testing	This testing is derived from creating threat trees from the threats identified in the threat model and ensures that the mitigations designed and implemented in the product are effective in stopping the proposed threat. Testers will design their tests to attempt to thwart the mitigation using the type of threat identified.
SV-3: General vulnerability testing	This testing focuses on using standard tools or published instructions for discovering potential security vulnerabilities. No attempt is made to exploit the vulnerability or assess the ability to exploit the potential vulnerability and the product is tested without consideration to the implementation or its defence in depth design.
SV-4: Penetration testing	This testing focuses specifically on compromising the confidentiality, integrity or availability of the product. It can involve defeating multiple aspects of the defence in depth design. This is an unstructured test that depends on the skills and knowledge of the attacker. In this case, the tester tries to play the role of an attacker. This testing is not based on an analysis of the design or threat model, rather it encompasses the tester trying to defeat the security of the system using any technique that he chooses. This testing often will identify types of vulnerabilities that need to be fixed rather than single vulnerabilities. This testing will often detect problems that are not detected in threat model driven testing because there may be errors or omissions in the threat model itself.

The testing has the following goals:

- Demonstrate that the defined secure configuration is correctly implemented.
- Demonstrate that the inner components are not influenced by an attacker.
- Demonstrate that the availability of the system is not compromised by an attacker.
- Demonstrate that the safety function of the system is not influenced by an attacker.

2.2.2 Subway pilot testing requirements

The high-level assumptions for the subway pilot follow from Czech national act No. 181/2014 on cyber security [13]:

- restriction of physical access to networks and equipment of IACS,
- restriction of interconnections and remote access to networks of IACS,

- protection of individual technological assets of IACS against attacks exploiting known vulnerabilities,
- restoration of the operation of IACS after cyber security incidents.

Assurance level, depth and scope of testing depend on demands and specific requirements of customers. UniControls has identified that the following common assumptions need to be verified:

1) Disabling all unnecessary ports.
2) Data encryption in open transmission system (category 3 according to[18]).
3) Data encryption for selected important information also in closed transmission systems [18] (e.g. video or important operation records).
4) Network segmentation (e.g. firewalls, VLAN, physical separation, etc.).
5) Access control options (rights) of user interfaces according to established security policies.

The security verification is mostly based on scanning for open ports, penetration tests and review of the actual implementation against documentation (e.g. block diagrams) that describes the security assumptions (e.g. list of ports, communication paths, interaction between users – including applications, etc.).

More and more, customers define the security assumptions using the set of requirements provided by IEC 62443[5], i.e. demands on security level and appropriate requirements from 62443-parts 3-3, 4-2 and occasionally part 2-4. Therefore, testing methods and requirements described in chapter 2.1, 2.4.1.3, IEC 62443 part 4-1 chapter 12 (code verification) and chapter 13 (requirements on security integration testing: test plan, fuzz testing and abuse case test) are practically increasing their importance. They will be necessary for security evaluation and certification of both compositional types to be described in chapter 2.4.

2.3 Embedded Device Security Assurance testing requirements

The IEC 62443 introduced in the previous section is a series of standards covering topics in the area of cybersecurity robustness and resilience in IACS of all industries. The series is organized into four groups addressing: general topics, policies and procedures, the system level and the component level. The latter two aspects can be certified by accredited labs using the ISASecure certification schemes.

2.3.1 What is tested?

ISASecure 62443 conformance certification has developed schemes for

Systems of full Industrial Automation and Control Systems (IACS), called System Security Assurance (SSA), and

Components of IACS, called Embedded Device Security Assurance (EDSA).

This distinction mirrors IEC 62443, where IEC 62443-4-* treat components of IACS and IEC 62443-3-* treat whole IACS systems.

Here we focus on EDSA, but also, for context and comparison, also partially treat SSA.

2.3.2 What is the testing for?

For IACS components, the testing is for confidentiality, integrity, and availability. For whole IACS, the testing is also for confidentiality, integrity and availability, but more specifically taking into account network stress testing.

During availability testing, IACS components need to maintain their essential functions. An essential function for an IACS component is (EDSA-310):

Downward: the control function, the process control loop, the safety instrumented function.

Upward: process view, command (meaning change parameters of process control such as set points), process alarms, peer-to-peer control communication. Providing process history is an essential function unless explicitly excluded by the certification applicant.

2.3.3 What are the methods?

In the following table (Table 4), testing guidance can be found detailed in three kinds of documents.

Table 4: ISASecure documents for robustness testing, functional requirements assessment, and life-cycle assessment.

	EDSA	**SSA**
Robustness (SRT)	Software robustness testing (SRT) [19] Ethernet testing: [20] ARP testing: [21] IPv4 testing: [22] ICMP testing: [23] UDP testing: [24] TCP testing: [25]	"System robustness testing" (SRT) [26]: has three major elements: 1. Vulnerability Identification Testing (VIT), 2. Communication Robustness Testing (CRT), and Network Stress Testing (NST) (SSA-300), 3. Asset Detection Testing (ADT). Nessus configuration to carry out VIT: [27].
Functional requirements (FSA)	Functional Security Assessment (FSA) [28]	"Functional security assessment for systems" (FSA-S), [29] For "Functional security assessment for embedded devices", called "FSA-E", it is pointed to EDSA-311. "*In particular, if a component of a system is a certified ISASecure EDSA embedded device, then FSA-E and the CRT aspect of SRT need not be performed on that device as part of the SSA certification process. This is due to the fact that these assessments will have been performed previously under the ISASecure EDSA certification process.*" [30]
Life-cycle assessment (SDLA)	Software Development Security Assessment (SDSA) [31]: points to SDLA, that is [32]	"Security development artefacts for systems" (SDA-S), [33]: points to SDLA, that is [32]

As, document-wise, ISASecure puts much emphasis on robustness testing, we start out with this aspect.

2.3.3.1 Robustness testing

Table 5: Robustness testing requirements and tools.

Characterization		Required for		Tools (see also D1.2)
		EDSA	SSA	
CRT	CRT examines the capability of the device to adequately maintain essential functions while being subjected to normal and erroneous network protocol traffic at normal to extremely high traffic rates (flood conditions). [34] CRT provides a measure of the extent to which IP-based protocol implementations defend themselves against • correctly formed messages and sequences of such messages; • single erroneous messages; and • inappropriate sequences of messages; [19] Section 1.2 For SSA, ADT (Asset Discovery Testing, i.e. port scanning, e.g. by nmap) is sometimes treated separately from CRT, for EDSA, ADT is referred to as “interface surface test” and always treated together with CRT.	yes	yes	e.g. Achilles, see ISASecure CRT Test Tools
NST	Network stress testing: apply CRT testing at high loads and observe that essential functions are maintained. [35] Section 6.3.5 gives an example on how CRT, NST, and VIT are applied in a concrete example system.	no	yes	Same as for CRT.
VIT	Vulnerability identification testing: VIT scans the device for the presence of known vulnerabilities.	yes	yes	e.g. Nessus, see ISASecure CRT Test Tools

2.3.3.1.1 Embedded device Robustness testing (ERT) in EDSA

Core protocols are ICMP, IPv4, ARP, IEEE 802.3, UDP or TCP (EDSA-310, Section 3.1). ERT has passed if:

- (CRT) essential functions are maintained under UDP and TCP port scanning and robustness testing of the core protocols (EDSA-312, ERT.R6 and ERT.R32), and
- VIT testing has not discovered any “Critical” or “High” risk factors (EDSA-312, ERT.R54). Nessus shall be used (EDSA-312, ERT.R50).

The protocol-specific CRT tests are specified at detail, we give two examples for the device under test (DUT):

- “Ethernet”.T02: IEEE 802.2 Type 1 with IEEE 802 SNAP misplaced Q-tag tolerance: The DUT SHALL protect itself against receipt of an IEEE 802 SNAP header that contains a Q-tag. [20]

- Load stress testing of the DUT's IPv4 implementation SHOULD include NPDU sequences that activate or cause error sequencing in any one of multiple concurrently-operating NPDU reassembly FSMs, including attempts to overload the state management capabilities of the DUT's IPv4 implementation. Requirement IPv4.R16 – Concurrent activation of multiple IPv4 FSMs for reassembling fragmented NPDUs: [22]

For VIT testing, the document [27] described how to configure Nessus.

2.3.3.2 Functional requirements

The way, functional requirements implemented by the system under test (SUT) are verified, depends on the claim. It can be *analysis* or *testing*.

Example where the validation activity is *analysis* (SSA-311 FSA-S-RDF-4 Application Partitioning):

Requirement: The SUT shall provide the capability to support partitioning of data, applications and services based on criticality to facilitate implementing a zoning model.

Validation: Verify SUT user documents include evidence that Application Partitioning capability is included to support zoning models and record results as:

a) Supported, or
b) Not Supported.

Example where the validation activity is *testing* (SSA-311 FSA-S-UC-4.3 Restricting mobile code transfer to/from the SUT):

Requirement: Restricting mobile code transfer to/from the SUT: The SUT shall provide the capability to enforce usage restrictions for mobile code technologies based on the potential to cause damage to the SUT that include restricting mobile code transfer to/from the SUT.

Validation: Connect a device to the SUT that contains mobile code not authorized to transfer to the SUT. Verify that the transfer is prevented and the user is notified of this occurrence. Record the results as:

a) Supported,
b) Not Supported,
c) Not applicable, if the device does not allow any mobile code to execute.

Note that EDSA-311, which is older than SSA-311, has similar requirements as SSA-311, but does not currently describe verification activities.

Example from *analysis* (EDSA-311):

FSA-RDF-3 Security Function Isolation. The IACS embedded device shall isolate security functions from non-security functions by means of partitions, domains, etc., including control of access to and integrity of, the hardware, software, and firmware that perform those security functions.

2.3.3.3 Lifecycle assessment based testing requirements

The security development lifecycle assessment is defined in SDLA-312 [32]. The document indicates the sources of the lifecycle based testing requirements, a collection of best practices from IEC 62443, ISO 61508, DO 178, Common Criteria for Information Technology Security, Comprehensive, Lightweight Application Security Process (CLASP), and Microsoft Secure Development Lifecycle (MS-SDL). Again, like in the SSA documents, validation activities are described. The description of validation activities is split into two types:

a) Validation of the development process itself.
b) Validation of the output of the applied development process to a certain component or system under test.

Example SDLA-SIT-1.2, Automatically Generated Test Cases, motivated from MS-SDL:

Requirement: The files or packets that will be used for fuzz testing ("fuzzed") shall be automatically generated so that a large number of test case (in the thousands) can be executed.

Component or system validation activity: Review fuzz test results and confirm that a large number of test cases were executed.

Development and SDL validation activity: Verify that the development process states that the files or packets that will be fuzzed shall be automatically generated so that a large number of test case (in the thousands) can be executed. Or, pick a product that is developed using the process under evaluation and review fuzz test results and confirm that a large number of test cases were executed. Note: Automated tools (e.g. Codenomicon Defensics, Hitachi Raven ES) are commercially available for certain types of fuzz testing.

2.3.4 What is the assurance gained?

EDSA / SSA CRT testing gives detailed robustness testing protocols for Ethernet and IP-based network protocols.

However, for robustness, it is also recognized that the goals of the CRT approach are limited *"to identify the presence of common programming errors and known denial of service vulnerabilities specifically for networking protocols, which impact the robustness of embedded devices that use these protocols. Tests are specified to a level such that these goals are covered, although specific test data is not defined. These tests will not necessarily identify intentionally malicious code, nor is that a feasible goal for any practical testing regimen."* [19].

Hence, it is up to FSA and SDLA testing *and* analyses to establish trust for the system under test.

2.4 Security testing and component composition

This chapter will look at security testing in a composed system. First the general use case in a layered MILS system, especially the I-composition, is explained. Then the approach is taken to the architecture used in the certMILS project pilots, with hardware, separation kernel and an application. The separation kernel is the hypervisor used in certMILS for safety certified systems. In this context, the hardware component is the sum of the physical platform and the specific platform support package (PSP), which is the hardware abstraction layer for the hypervisor used in certMILS.

2.4.1 Typical general use-case

In a composed system, a threat analysis generally models an attacker and assets. Figure 4 shows the typical setup in a layered composed system, which has an outer and an inner component. Both the inner and the outer component potentially contain assets. In the example, the outer component covers the inner component. The outer component interacts with the attacker, and communicates, if the request is well-formed, and asks for a resource provided by the inner component, with the inner component.

- The API of the outer component has to be robust against malicious inputs.
- The API of the inner component does not have to be robust against malicious inputs, if the outer component successfully filters out malicious inputs. That is the inner component relies on that the outer component filters certain input (upper two arrows). Conversely, the outer component can rely on that the flow from the inner component to satisfy the specification of the inner component.

2.4.1.1 What is tested?

In this scenario, the interface to the outer component is tested.

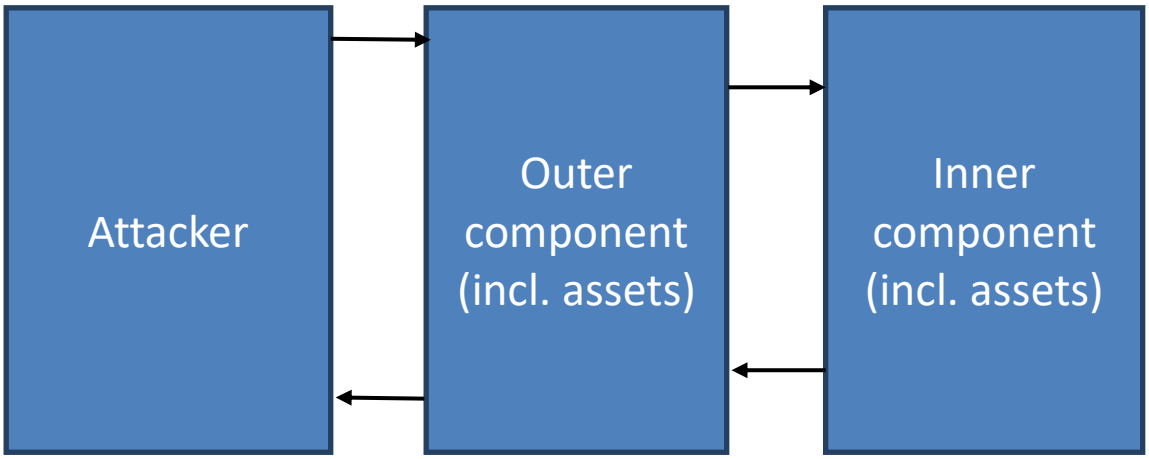

Figure 4: Location of attacker in a layered composed system.

2.4.1.2 What is the testing for?

The testing has the following goals:

- Test that the outer component and its assets, if any, cannot be compromised by the attacker.
- Test that the inner component and its assets, if any, cannot be compromised by the attacker.
- Test that the outer component shields all non-robust, if any, parts of the inner API against an attacker. Specifically, the outer component guarantees that the inner component cannot be attacked by passed-on input data.
- Test that the attacker cannot use the inner component to bypass the outer component. Specifically, the inner component guarantees that its output does not attack the outer component.

2.4.1.3 What are the methods?

Deliverable D1.2, Section 1 (in preparation) gave a table of testing methods according 62443-4-1, Section 10.1. For reference, that information can also be found in Table 3.

We now apply the classification of Table 3 to the scenario of testing inner and outer components, when one is directly acting as attacker. The results are shown in Table 6.

Table 6: Application of testing methods to a composed system.

Name	Outer component	Inner component
SV-1: Security requirements testing	Feasible.	Can only be done for security requirements exported to the outer component.
SV-2: Threat mitigation testing	Threats cover assets provided by outer and inner component.	Threats cover assets provided by inner component.
SV-3: General vulnerability testing	COTS tools can be used be for network protocols etc. Fuzzing is needed for more product-specific interfaces.	Tools likely not available.
SV-3-x: Specifically, fuzzing	Fuzzing needed for more product-specific interfaces. API fuzzers use the regular interfaces into a subsystem. In-memory fuzzers can inject data into a system by directly manipulating stack frames, registers or other memory locations. NB: Probably not very interesting for the separation kernel used in certMILS as we explicitly want to test the APIs, handle with low priority	Needs specialized fuzzer architecture ("layered fuzzing"), adapted to often non-standardized inner component. Successful coverage-based layered fuzzing, which starts at the outer interface, might penetrate to the interface of inner components.
SV-4: Penetration testing	Based on analysis of the external interface to the attacker.	Based on analysis of the internal interface and how / what parts of it are exposed to the attacker.

2.4.1.4 What is the assurance gained?

The gained assurance mirrors the testing's target, as mentioned in Section 2.4.1.2.

The testing has the following goals:

- Demonstrate that the outer component and its assets, if any, cannot be compromised by the attacker.
- Demonstrate that the inner component and its assets, if any, cannot be compromised by the attacker
- Demonstrate that the outer component shields all non-robust (if any) parts of the inner API against an attacker. Specifically, the outer component guarantees that the inner component cannot be attacked by the input not filtered.
- Demonstrate that the attacker cannot use the inner component to bypass the outer component. Specifically, the inner component guarantees that its output does not attack the outer component

More generally, regardless of the testing method, give convincing evidence that compositional aspects have been taken into account.

2.4.2 Separation kernel I-composition: PSP

The I-composition concept is described in detail in D1.1[13]. The PSP component decouples all other separation kernel components from platform or board specific details at source and object code level, which means that the PSP can be exchanged without recompiling any of the other components.

The main tasks of the PSP are:

- Platform initialization
- Interrupt management
- Hardware timer management
- Console output support
- Memory region management

During project build, the PSP is linked to the kernel and becomes a part of it. The PSP implements a number of entry points, which to be called by the kernel.

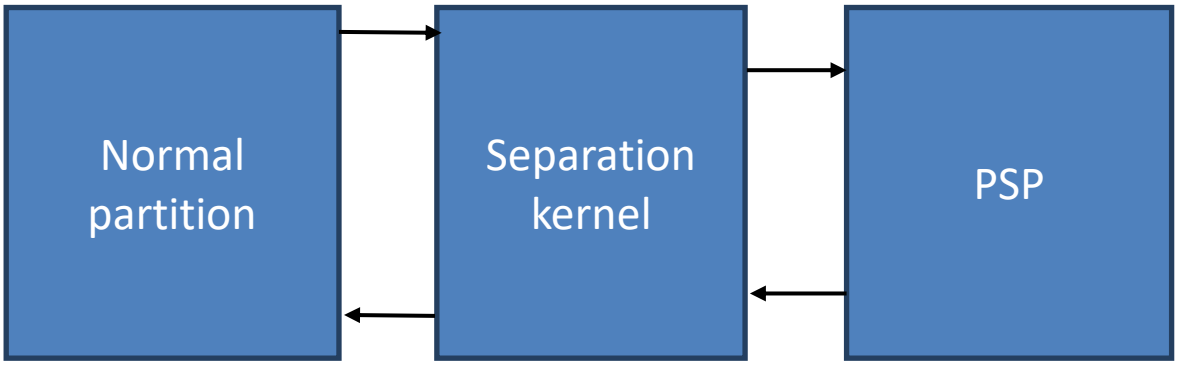

Figure 5: PSP services as part of system composition.

For a separation kernel according to ST (Deliverable D3.1, in preparation) and PP (D2.1, in preparation) attackers are executables in normal partitions. Thus, if we instantiate Figure 4 for the PSP, then we obtain the situation shown in Figure 5 for the PSP.

Assets maintained by the PSP are the services provided by the PSP: memory and interrupts (see Deliverable D3.2). The PSP governs resources of interrupts and memory. They are provisioned by the PSP at initialization time, giving rise to a variation in the control flow, as shown in Figure 6. Here the run-time permissions by a normal partition might include load/store accesses to memory, reading out the interrupts and hardware timer.

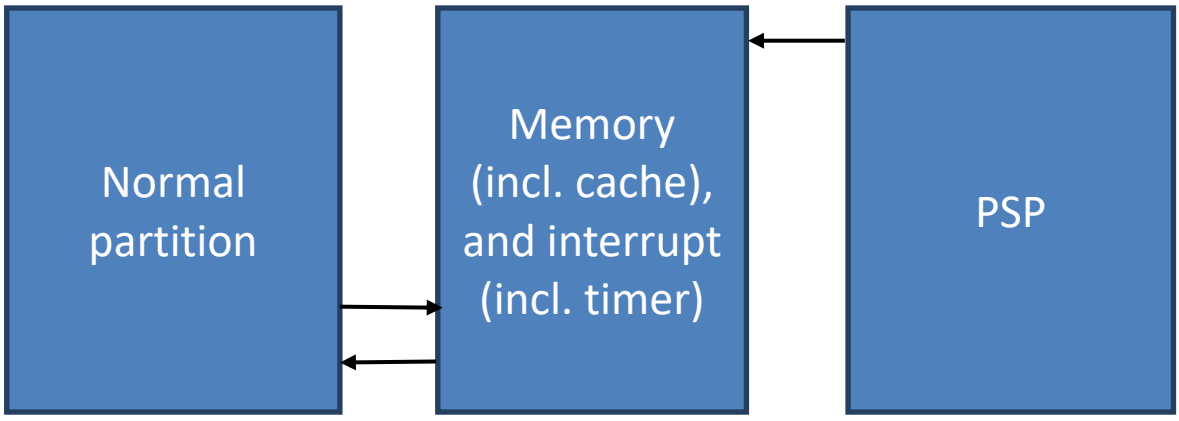

Figure 6: PSP resource management.

An interesting special case is a composition with an I/O MMU, represented by Figure 7:

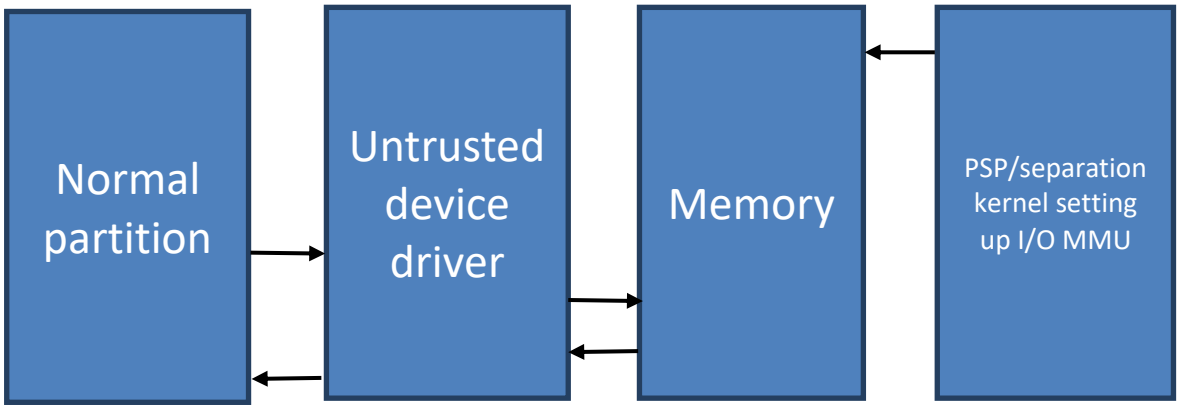

Figure 7: Composition in context of an I/O MMU driver.

That is, a PSP, part of the separation kernel, sets up an I/O MMU at initialization time, so that DMA accesses from an untrusted device driver are controlled. As a further indirection layer, that untrusted device driver is controlled from a normal partition.

2.4.3 Separation kernel I-composition: kernel device driver API

Kernel device drivers are executed in the separation kernel's microkernel context within the Kernel Driver framework. They support the Port API and the File API, which are comparable to the POSIX file API. Kernel device drivers are directly linked to the kernel object code.

2.4.3.1 Description of scope and implementation

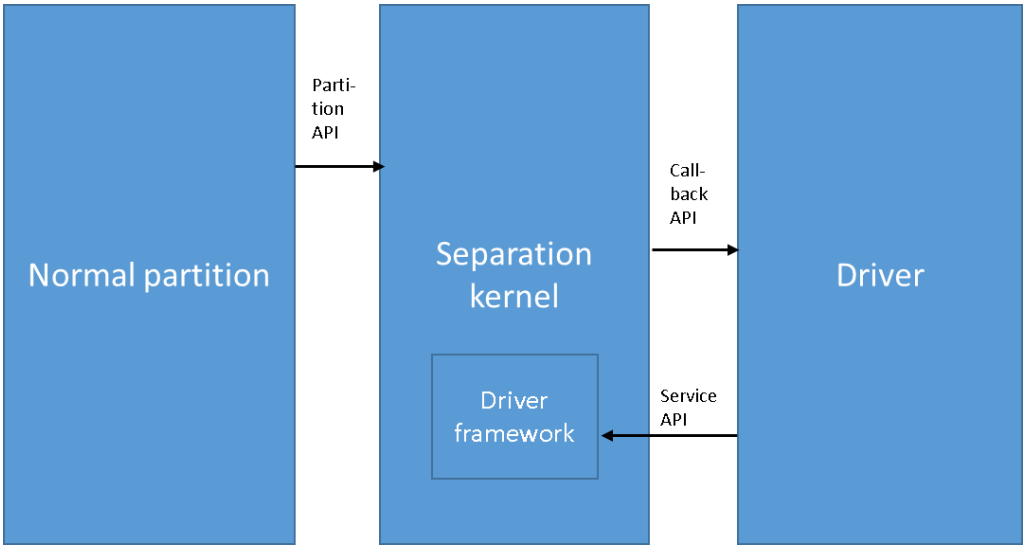

Figure 8: Correlation of the APIs for kernel device drivers.

We describe three APIs, which can directly or indirectly, interact with kernel device drivers.

1) The first API (called "Partition API" in Figure 8) to kernel device drivers is part of the Partition API, which is the system call. It allows to control the driver from within a resource partition. This API is robust (see API classification in section 3.3).
2) The second (inner API) API is the "Callback API". Each kernel device driver must register a certain set of callbacks, which will be executed by the kernel for example as a result of interaction from user space. The inner API relies on guaranteed filtering by the outer

"Partition API". In return, the inner API guarantees that its output is according to specification.

3) The "Service API" contains functionality that the kernel device driver framework provides to the drivers to implement their functionality. It consists of more than 100 functions for operations like data transfer from/to user space, locking, interrupt management and others. The "Service API" requires that kernel device drivers are implemented correctly.

2.4.4 T-composition scenarios

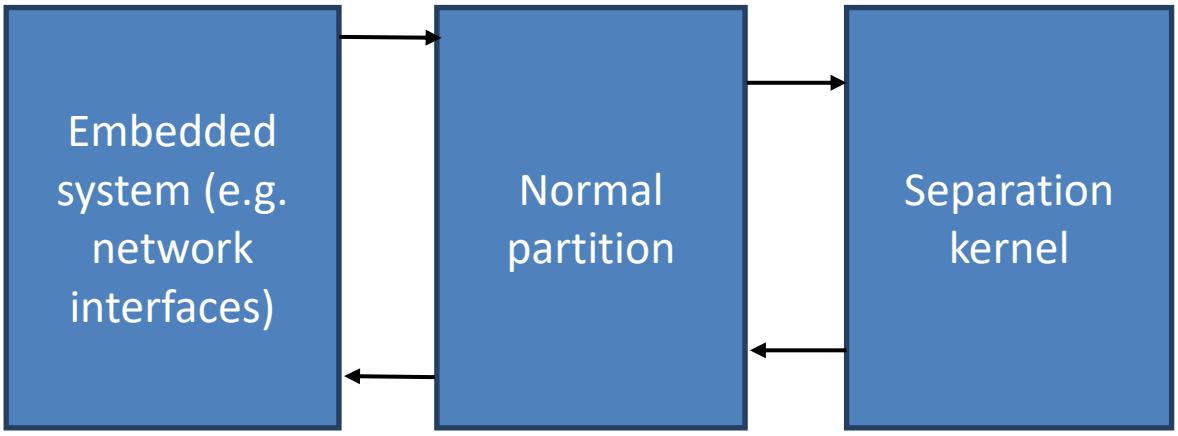

Figure 9: T-Composition.

Finally, there is the T-composition (see D1.1[13] Section 5.4.1), whose compositional layout is shown in Figure 9. In such a scenario, a vendor, who exposes e.g. an Ethernet network interface, wants to understand to what extent that would allow attacks on the separation kernel, or partitions hosted by it.

- If the Ethernet network interface is implemented in a *non-privileged partition*, this partition cannot attack applications in other partitions, thanks to assurance of the separation kernel. The only attack vector is to deny services or alter the content of allowed communication from the Ethernet network interface partition with other partitions.
- If the Ethernet network interface is implemented in a *privileged partition*, then that privileged partition can try to attack applications in other partitions or even the separation kernel itself.

Hence, in a T-composition scenario, if an application is in a privileged partition the system needs to be analysed/tested in its entirety. If the application is in an untrusted unprivileged (normal) partition, then it suffices to analyse/test the extent (if any at all) the application can affect other partitions by allowed interaction of the partition with other partitions.

2.4.5 Compositional testing in the pilots

The T-composition scenarios in the subway pilot are an implementation of the MILS platform – based on the I-composition as a generic system – at the application level intended to the specific use in subway operations. Because the T-composition is an implementation for the specific use, it needs to be tested against specific requirements, including security requirements.

The testing framework also needs to consider specific applications for an I-composition to provide a wide range of benefits. Otherwise, it could result in increased development and testing efforts, increasing product costs and no advantage from security certification due to different security targets. The final system implementation in specific use shall, as much as possible, exploit features

of the I-composition and profit from them (i.e. improve cyber security of designed system, decrease development effort and costs).

Examples of benefits are as follows:

- layer of Defence-in-Depth approach that supports and multiplies system level protection,
- higher level of security assurance for selected security functions,
- decreased system development costs through application of security functions by system integrators for improved system security.

Current practice for security verification (including testing) of system with specific use follows from process of risk analysis[36], mainly:

- risk identification (assets, threats, *existing measures*, vulnerabilities),
- risk analysis and assessment (impacts, likelihoods, risk level, assessment),
- risk treatment (modification, applying, avoiding, transferring)

Monitoring and review are part of farther phases of the system lifecycle. Therefore, they are not the base for the definition of test cases. The risk analysis is mostly performed using a risk table that covers all of the mentioned security aspects (assets, threats, vulnerabilities, impacts, likelihoods, risks and measures, etc.).

The system environment, existing measures or designed countermeasures within the process of risk treatment create a set of *security assumptions*. The security assumptions shall be verified and tested, if possible. Because changes in these can lead to new risks and invalidate the previous security risk table. These new risks shall be again analysed and assessed.

Chapter 3 Separation kernel security fuzzing

This section discusses security fuzzing of separation kernels in a generic way. To develop a fuzzing strategy, it is important to have a clear understanding of interfaces. A separation kernel either can be monolithic or have a microkernel. To realize a separation kernel, often microkernel designs (e.g. [37][38][39][40]) have been applied. In a separation kernel based on a microkernel:

- The *microkernel* implements only the most fundamental operating system services, including protection of memory and access to CPU privileged states, in a privileged domain [41].
- Less critical services are in one or several non-privileged domains, called "*system software*" (SSW).

Other separation kernels do *not* use or at least do not claim a microkernel design, e.g. previous protection profiles for separation kernels [42][43] were architecture-agnostic. In terms of interfaces, a monolithic kernel can be seen as a specific microkernel without user-space system software. So to cover the more general case, we assume that the separation kernel has a microkernel architecture.

3.1 Motivation

As described in Figure 2, security testing consists of

1) SFR / functional testing ,
2) penetration testing,
3) static analysis,
4) fuzzing.

Items (1) and (2) from the above list are already covered by the CC evaluation in certMILS Task 5.5. Here we concentrate on fuzzing. In safety, "penetration testing" is also referred to as "robustness testing".

3.2 Preamble to separation kernel fuzzing

This section is work towards an improved separation kernel fuzz testing strategy. Its purpose is to:

- Give an overview of separation kernel interfaces and existing fuzzing techniques that could be applied to them by cross matching in a matrix.
- Roughly, estimate the applicability, technical challenges and benefits that will arise in certain combinations of interfaces and techniques.

3.3 Separation kernel interfaces

Currently, four separate interfaces (APIs) to a separation kernel's microkernel and its system software (see overview in Figure 10) are considered candidates for fuzzing. These are briefly discussed here, based on information sourced from the separation kernel's user documentation.

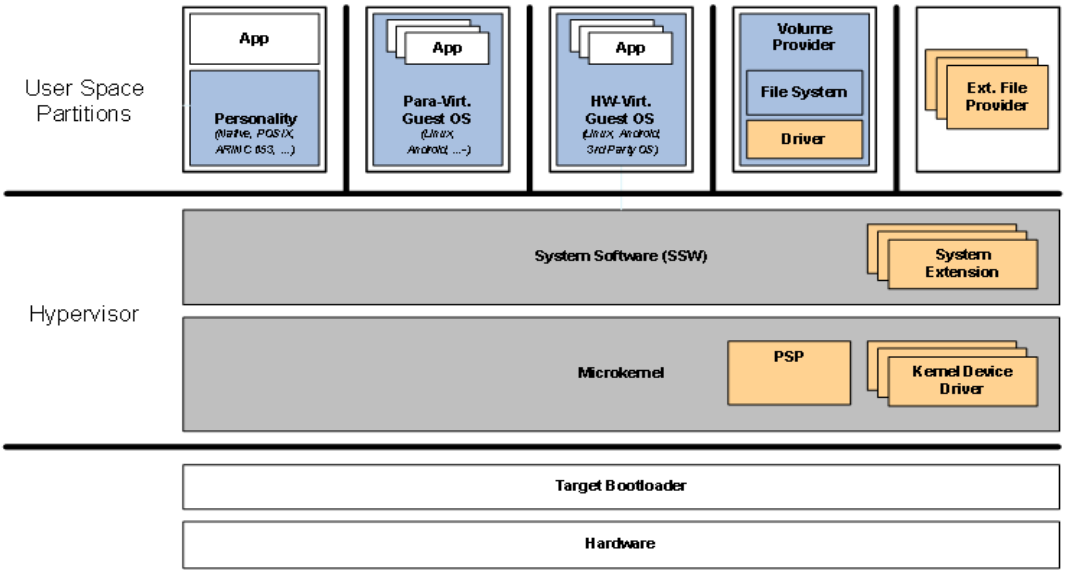

Figure 10: Separation kernel system architecture with highlight on User/Kernel mode scope.

Separation kernel APIs can be separated into two groups, which are categorized as *robust* and *non-robust* here:

Robust APIs occur where callee and caller have different trust/integrity levels, i.e. code with higher trust is called by code with lower trust. This includes potentially malicious code. No conceivable call or sequence of calls should be able to compromise the integrity or availability of a robust API. In terms of fuzzing, every anomaly uncovered here needs investigation.

Non-robust APIs occur between application parts within in the same security domain, i.e. they trust each other with regards to input and output – e.g. they share the same address space and can interfere with each other freely. Since they "trust" each other, the API does not need to guarantee that all arguments are completely sanitized. Fuzzing non-robust APIs has to be done with greater care, since it is possible to "abuse" the API and get false positive findings as a result.

Note: the protections that have to be implemented for APIs that trust the callers (non-robust APIs) are not that high as the protections that have to be implemented for APIs that does not trust the callers An issue here comes when the caller which is assumed to be trusted is not trusted anymore (e.g. when it has been attacked/hijacked). During the penetration testing phase this scenario must covered.

3.3.1 Microkernel API

The microkernel API is the main interface from user space into the separation kernel's microkernel. This does also mean a transition of the CPU from user mode into kernel mode (system call).

Both the applications within resource partitions as well as the SSW use the microkernel API to communicate with the separation kernel's microkernel. However, the SSW has a higher privilege level and can therefore use API calls that normal partitions cannot use. This is controlled through permissions ("abilities"). The mapping of system calls ("syscalls") to abilities is listed in the

documentation. The SSW task automatically has all existing abilities. Some abilities are also SSW-exclusive and cannot be granted to tasks in regular partitions.

The microkernel API is exposed to normal partitions.

Provided services in the microkernel API are:

- Hardware abstraction,
- Resource and time partitioning,
- Execution entities (threads),
- Separate address spaces (tasks),
- Communication primitives,
- Timers,
- Exception and interrupt handling,
- Health Monitoring.

Description of scope and implementation

The API currently consists of “true” syscalls, i.e. calls that enter the kernel.

3.3.2 System Software API

The System Software (SSW) component is the first user space application launched by the microkernel. The SSW component reads the configuration and initializes partitioning and inter-partition communication according to the configuration. During run-time the system software component acts as a server providing the following services to the applications executing inside the different resource partitions:

- Communication via queuing and sampling ports,
- File system services,
- Partition and process management services.

Communication with the SSW takes place via Inter-Process Communication (IPC). Parameters for the SSW API calls are serialized into the IPC payload by a user space library and de-serialized by the SSW (return values vice versa).

Like the microkernel API, the SSW API is considered *robust*.

Description of scope and implementation

The SSW code is executed by so-called partition daemons, which are automatically created for each partition. Communication with these daemons works via IPC messages. This has the following implications:

- All calls to the SSW actually pass through the kernel.
- The SSW service calls are wrappers that serialize the arguments to the SSW into IPC messages, which un-wraps them on the other side by decoding the command code and the arguments.
- From the security view, attackers can bypass the SSW service call wrappers and all argument sanitation they might perform. A “security fuzzer” should therefore interface directly via the IPC interface with the SSW.
- From the safety point of view, the application would always use the SSW functions. A “safety fuzzer” should therefore use the wrappers.

3.3.3 Kernel device driver API

See the previous chapter 2.4.3 Separation kernel I-composition: kernel device driver API.

3.3.4 Platform Support Package (PSP) API

See the previous chapter 2.4.2 Separation kernel I-composition: PSP.

3.4 Fuzzing aspects

This section briefly describes different approaches to fuzzing that could be applied to the separation kernel interfaces that have been described in the previous section. The taxonomy partially follows the source in[4]. The aspects are mostly orthogonal, so that every fuzzer will have one or more aspects from each group.

3.4.1 Input generation

Brute-Force based fuzzers generate input completely at random.

Mutation-based fuzzers start with valid input and mutate it by introducing more or less subtle changes. NB: A previous project tested mutation-based fuzzing with no satisfying results. This technique should not be researched with high priority.

Template-based fuzzers follow a pre-defined ruleset in order to generate more “interesting” data that can get past input sanitation.

Coverage-guided fuzzers are able to extract code coverage information from the code under test and can use this information to adapt the input in order to maximize coverage.

3.4.2 Parallelism

Single-threaded fuzzers are suited to uncover logic bugs such as off-by-one errors which occur deterministically.

Multi-threaded fuzzers can additionally trigger concurrency bugs such as deadlocks and race conditions. This type of bug is typically non-deterministic and depends on random events such as small timing deviations.

3.4.3 State

Stateless fuzzers perform every iteration independently from the previous ones.

Stateful fuzzers preserve state information (e.g. open file handles) between iterations.

3.4.4 Adherence to subsystem boundaries

API fuzzers use the regular interfaces of a subsystem.

In-memory fuzzers can inject data into a system by directly manipulating stack frames, registers or other memory locations. NB: Probably not very interesting for the separation kernel as we explicitly want to test the APIs.

3.4.5 Layered fuzzing

The PSP / kernel device driver can be indirectly fuzzed from user-space.

A related approach, to apply in-system fuzzing is published through the “kAFL – OS kernel fuzzing” framework[44]. The implementation constructs a fuzz-agent, that is injected into the system under test and tests the test case’s API directly where it has the required access. In terms of the separation kernel and in-kernel API, this could be a test-specific kernel-driver.

3.5 Cross-matching matrix

Below, in Table 7, an estimate of the applicability, technical challenges and benefits to matched to separation kernels interfaces and fuzzing aspects is given.

Table 7: Fuzzer property and applicability cross-matching matrix.

Aspect/API	Robust APIs (e.g. Microkernel or SSW interfaces)
Brute-force	Sometimes brute-force fuzzing brings early (sometimes not very clear though) results that can be later used in most advanced fuzzing techniques. Implementing a brute-force fuzzer is not complex, and the most timing consuming phase for brute-force fuzzers is the execution, which does not need to be human attended. As a first step, brute-force fuzzing should always be taken into account for "Robust APIs". Even if the results might be useless, it brings a clear scenario for further fuzzing instead.
Template-based	This can be done by selecting the arguments for kernel APIs are selected from the valid class of arguments with a special focus on the boundary values.
Coverage-based	If the separation kernel already has tool support for extracting coverage information as required by standards such as DO-178 [45] (e.g. Rapita tool suite), then this is possibly easy to complement. If the format of coverage data is compatible to the GCC gcov format, then the syzkaller Linux fuzzer could be re-used for separation kernel fuzzing.
Multi-Threaded	Currently implemented: “During the test, the system state is concurrently modified by high priority threads to induce race conditions.”
Stateful	State awareness requires template knowledge
Layered fuzzing	Not applicable

Aspect/API	Non-robust APIs (e.g. kernel device drivers, PSP)
Brute-force	Might be problematic for non-robust APIs that do not need to perform thorough input sanitization.
Template-based	Lower priority in comparison to coverage-based
Coverage-based	Intended, feasibility to be proven
Multi-Threaded	Optional, complex to realize in comparison to single-threaded
Stateful	State awareness requires template knowledge, or in combination with layered fuzzing and fuzz-agents
Layered fuzzing	Applicable

3.6 Hardware support and integration

Stateless and brute-force fuzzing do not depend on any feedback from the test target. This simplifies implementation but is very inefficient, as many inputs will be discarded by simple input check mechanisms of the test target and large dimensions of valid inputs may not reach critical parts.

Template-based fuzzer can leave out inputs that are irrelevant due to input checks, or which do not reach separate code paths. On the one hand, this can be very efficient to narrow down the input space. On the other hand, API designers or testers need to create these templates first, which is time consuming and might introduce template tainting. Template-based Linux-system-call fuzzer, such as Syzkaller, also use code coverage through the kcov-kernel interface to improve their progress.

Code coverage information improves progress to 'interesting' input corpuses drastically. For example, the coverage-feedback based general-purpose fuzzer AFL is known to craft valid (but meaningless) JPEG images through improving a random start input towards good coverage in a JPEG-library.

There are different approaches to collect coverage information from the target. The current solution used by AFL is to instrument branches and calls with extra assembler instructions that copy the code address taken to a shared memory segment. When the AFL fuzzer runs, it forks multiple instances of the target program, feeding generated and mutated inputs. The processes execute and may crash upon the particular input. The disadvantages for kernel fuzzing are the architecture dependent assembler instructions, the user space fork-to-crash approach and the focus on input-delivery via the standard input file. Related extensions to AFL have moved the code instrumentation into compiler-plugins to reduce the instrumentation overhead through C-compiler optimizations. Even smoother integration with the compilation process is achieved with libFuzz as part of the LLVM/Clang compiler toolchain.

However, the concurrent process-fork approach does not apply to kernel fuzzing. A system can only run a single kernel and a kernel-crash would always interrupt the fuzzing process and may corrupt feedback evaluation. Consequently, for efficiency the fuzzing process cannot run as a user process of the kernel under test. Thus, kernel fuzzing requires use of virtualization techniques.

In a virtualized scenario, the fuzzing framework starts the test-target in a virtualized environment, equipped with a fuzz-agent to inject input data and feedback coverage data. Schumilo et al. [44] published this approach as kAFL for COTS-kernels (Linux, Windows, IOS). To excel performance, kAFL uses a modified version of the virtualization environment QEMU-KVM. QEMU is an open-source full-system virtualization software. KVM is the "kernel-based virtual machine" for Linux operation systems for hardware-assisted virtualization of native guests (same target architecture). The modification provided by kAFL introduces support of the PT hardware tracing features (Intel Processor Trace) of current Intel x86 processors. An additional translation tool converts the captured trace data to AFL coverage maps and feeds it back to kAFL for the next invocation of the virtualized system.

The kAFL approach currently outperforms any other generic kernel fuzzing framework (e.g. TriforceAFL), while sustaining good flexibility. The authors have published their approach only for consumer OS (Mac, Windows, Linux). Nevertheless, there is strong confidence that this approach also applies to the separation kernel, as it also runs well in a QEMU virtualization environment.

The biggest challenge in porting the kAFL approach to certMILS is in adapting the trace collection features to other architectures used in certMILS, predominantly ARM64 and PowerPC. Both architectures have built-in debug and tracing support:

- NXP Nexus Trace in the e6500 PowerPC core (e.g. NXP T4080)
- ARM CoreSight for ETMv4 in cores A53 and A72 (e.g. NXP LS1043)

Linaro, as the leading maintainer of the open source ARM toolchain, recently added support for the Coresight infrastructure to the Linux kernel trace subsystem. In conjunction with QEMU and KVM this could enable efficient virtualized fuzz testing of the separation kernel and the PSP on the target system. Furthermore, support for Nexus Trace on PPC and alternatively user-land AFL-QEMU-mode performance need to be evaluated for feasibility and usability in MILS Security Testing.

The relevance of code coverage information also has a good correlation to potentially found bugs, as was demonstrated in [7] (chapter 8.8.6) in 2008. However, the authors make a distinction that coverage through a particular interface is often limited to subsections of the code. Adjusting for relative coverage shall not lead to false positive metrics. The security framework addresses this bug tainting through layered fuzzing techniques.

Chapter 4 Summary and conclusion

This deliverable has researched the state of the art in security testing techniques. The research was especially focused on techniques applicable for Industrial Automation and Control Systems with elevated assurance levels. The assurance requirements also include Information Security Assurance to raise confidence in the dependability of the system. Dependability is composed of measures of safety and security. Safety is the adequate reduction of risk that the system can harm its environment. However, safety is also tightly coupled to availability and integrity of the system to fulfil its safety policy, i.e. to correctly perform the safety function(s). Security techniques ensure availability and integrity by employing countermeasures to mitigate threats that impose risks on the assets.

Safety measures reduce risks coming from random and systematic failures, which are well modelled and understood, because they are under the control of the supplier. In contrast, security threats are imposed by the environment, which is uncertain and may change without control of the supplier, thus potentially invalidating static countermeasures against known vulnerabilities.

These challenges are addressed by standards for testing contexts. This deliverable has analysed the relevant standards CC and IEC 62443, as well as the related certification scheme ISASecure/EDSA coupled to IEC 62443 and derived testing methods for component compositions. Some of the resulting requirements are addressed in other deliverables (e.g. existing test tools in D1.2).

One of the currently most prominent testing techniques discovering hard to find vulnerabilities is fuzz testing. Due to the complex mechanisms to apply fuzz testing to a ToE, fuzz frameworks need to be developed or derived from generic approaches refined for the ToE. In a MILS system this are components or compositions thereof, involving user-space applications, OS and kernel components and HW coupled components (e.g. PSP). These aspects require different technical approaches, which were analysed in a separate chapter. It concludes with the evaluation of a recently published approach to effectively apply fuzz testing to general purpose operating systems on x86-architecture platforms. Since the hardware architectures in the certMILS project differ, i.e. PowerPC and ARM, a different approach to extract processor-based code coverage information is required. Initial research has found that recent processors of these architectures also provide extended tracing technologies for in-system analysis. As a result, the certMILS project will subsequently evaluate the technical feasibility for the security.

Hardware assisted fuzz testing is an important feature of the security testing framework, to provide robustness tests for most of the components of the MILS systems. This testing can continuously demonstrate high level of robustness, resulting in confidence that composed systems, e.g. with patched vulnerabilities, still uphold their effective security measures. This in turn results in integrity and availability of the system to enforce the required safety policy. Effective security measures of the basic component providing separation are the basis for a certifiable system, e.g. in a layered composition, where an inner component executing a safety function has reduced security requirements due to a greatly reduced attack surface.

As hinted before, one of the utmost expectations on the security framework is its availability not just to the component developer, but also to the system supplier for integration testing, as well as the system operator to verify operational patches for the complete lifecycle. Accompanying guidance to dissect the results of testing framework, especially the fuzzers, must be accessible and helpful for all technical roles. These requirements and the long term availability have consequences on the choice of COTS and customized tools. The refinement of the interpretation parameters of the guidance tools will depend on the output of the test tools, thus will require further analysis.

Chapter 5 List of abbreviations

Abbreviation	Translation
ADT	Asset Discovery Testing
ARP	Address Resolution Protocol
ATE	CC Assurance class: Tests
AVA	CC Assurance class: Vulnerability Assessment
CC	Common Criteria
CLASP	Comprehensive, Lightweight Application Security Process
CRT	Communication Robustness Testing
CVE	Common Vulnerabilities and Exposures
DUT	Device under Test
EAL	Evaluation Assurance Level
EDSA	Embedded Device Security Assurance
ERT	Embedded Device Robustness Testing
FSA	Functional Security Assessment
FSM	Finite State Machine
IACS	Industrial Automation and Control Systems
ICMP	Internet Control Message Protocol
IPC	Inter-Process Communication
NPDU	Network Protocol Data Unit
NST	Network Stress Testing
OWASP	Open Web Application Security Project
PP	Protection Profile
PSP	Platform Support Package
SDL	Secure Development Lifecycle
SDLA	Security Development Lifecycle Assessment

Abbreviation	Translation
SDSA	Software Development Security Assessment
SecRS	Security Requirements Specification
SRT	System Robustness Testing
SSA	System Security Assurance
SSW	System Software
ST	Security Target
SUT	System under test
TCP	Transmission Control Protocol
ToE	Target of Evaluation
TSF	ToE Security Functionality
TSFI	ToE Security Functionality Interface
UDP	User Datagram Protocol
VIT	Vulnerability Identification Testing

Chapter 6 Literature

[1] The Open Web Application Security Project, “The OWASP Testing Framework,” [Online]. Available: https://www.owasp.org/index.php/The_OWASP_Testing_Framework. [Accessed September 2017].

[2] P. Godefroid, M. Y. Levin and D. Molnar, “SAGE: Whitebox Fuzzing for Security Testing,” *Commun. ACM,* pp. 40-44, March 2012.

[3] Google, “OSS-Fuzz - Continuous Fuzzing for Open Source Software,” [Online]. Available: https://github.com/google/oss-fuzz/. [Accessed August 2017].

[4] H. Pohl, “https://www.softscheck.com,” 2011. [Online]. Available: https://www.softscheck.com/publications/ProfDrHartmutPohl_Identifizierung_unbekannter_Sicherheitsluecken_und_Software-Fehler_durch_Fuzzing_kes20115.pdf. [Accessed 2017].

[5] IEC, “IEC TS 62443-1-1:2009 Industrial communication networks - Network and system security - Part 1-1: Terminology, concepts and models,” IEC, Geneva, 2009.

[6] P. Oehlert, “Violating assumptions with fuzzing,” *IEEE Security Privacy,* vol. 3, no. 2, pp. 58-62, 03 2005.

[7] A. Takanen, J. DeMott and C. Miller, Fuzzing for Software Security Testing and Quality Assurance, Norwood: Artech House Publishers, 2008.

[8] B. Shastry, F. Maggi, F. Yamaguchi, K. Rieck and J.-P. Seifert, “Static Exploration of Taint-Style Vulnerabilities Found by Fuzzing,” *arXiv.org: Cryptography and Security,* p. https://arxiv.org/abs/1706.00206, 2017.

[9] M. Zalewski, “american fuzzy lop,” [Online]. Available: http://lcamtuf.coredump.cx/afl/. [Accessed 2017].

[10] The LLVM Foundation, “libFuzzer – a library for coverage-guided fuzz testing,” [Online]. Available: http://llvm.org/docs/LibFuzzer.html. [Accessed 2017].

[11] P. Cordemans, J. Boydens and E. Steegmans, “Testing concurrent software: challenges and tools,” in *Belgium Testing Days*, Brussels, 2015.

[12] L. N. R. Kakulamarri and M. S. Musuvathi.US Patent US8533682 B2, 2010.

[13] certMILS, “D1.1 Regulative Baseline: Compositional Security Evaluation,” EC, 2017.

[14] CCMB, “Common Criteria for Information Technology Security Evaluation v3.1, Part 1: Introduction and general model,” 2017. [Online]. Available: https://www.commoncriteriaportal.org/files/ccfiles/CCPART1V3.1R5.pdf.

[15] CCMB, “Common Criteria for Information Technology Security Evaluation v3.1, Part 3: Security assurance requirements,” 2017. [Online]. Available: https://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R5.pdf.

[16] CCMB, “Common Methodology for Information Technology Security Evaluation v3.1,” 2017. [Online]. Available: https://www.commoncriteriaportal.org/files/ccfiles/CEMV3.1R5.pdf.

[17] The MITRE Corporation, “CVE - Common Vulnerabilities and Exposures,” 2017. [Online]. Available: http://cve.mitre.org/.

[18] CENELEC, “EN50159:2010 Railway application – communication, signalling and processing systems – safety-related communication in transmission systems,” CENELEC, Brussels, 2010.

[19] ISA Security Compliance Institute, “EDSA-310 Embedded Device Security Assurance – Requirements for embedded device robustness testing Version 2.2,” 2015. [Online]. Available: http://www.isasecure.org/en-US/Certification/IEC-62443-SDLA-Certification.

[20] ISA Security Compliance Institute, “EDSA-401 Embedded Device Security Assurance – Testing the robustness of implementations of two common "Ethernet" protocols Version 2.01,” 2010. [Online]. Available: http://www.isasecure.org/en-US/Certification/IEC-62443-SDLA-Certification.

[21] ISA Security Compliance Institute, “EDSA-402 Embedded Device Security Assurance – Testing the robustness of implementations of the IETF ARP protocol over IPv4 Version 2.31,” 2010. [Online]. Available: http://www.isasecure.org/en-US/Certification/IEC-62443-SDLA-Certification.

[22] ISA Security Compliance Institute, “EDSA-403 Embedded Device Security Assurance – Testing the robustness of implementations of the IETF IPv4 network protocol Version 1.6,” 2015. [Online]. Available: http://www.isasecure.org/en-US/Certification/IEC-62443-SDLA-Certification.

[23] ISA Security Compliance Institute, “EDSA-404 Embedded Device Security Assurance – Testing the robustness of implementations of the IETF ICMPv4 network protocol Version 1.3,” 2010. [Online]. Available: http://www.isasecure.org/en-US/Certification/IEC-62443-SDLA-Certification.

[24] ISA Security Compliance Institute, “EDSA-405 Embedded Device Security Assurance – Testing the robustness of implementations of the IETF UDP transport protocol over IPv4 or IPv6 Version 2.6,” 2010. [Online]. Available: http://www.isasecure.org/en-US/Certification/IEC-62443-SDLA-Certification.

[25] ISA Security Compliance Institute, “EDSA-406 Embedded Device Security Assurance – Testing the robustness of implementations of the IETF TCP transport protocol over IPv4 or IPv6 Version 2.01,” 2015. [Online]. Available: http://www.isasecure.org/en-US/Certification/IEC-62443-SDLA-Certification.

[26] ISA Security Compliance Institute, “SSA-310 System Security Assurance – Requirements for system robustness testing Version 2.0,” 2015. [Online]. Available: http://www.isasecure.org/en-US/Certification/IEC-62443-SSA-Certification.

[27] ISA Security Compliance Institute, “SSA-420 System Security Assurance – Vulnerability

Identification Testing Policy Specification Version 2.6," 2014. [Online]. Available: http://www.isasecure.org/en-US/Certification/IEC-62443-SSA-Certification.

[28] ISA Security Compliance Institute, "EDSA-311 Embedded Device Security Assurance - Functional Security Asssessment (FSA) Version 1.4," 2010. [Online]. Available: http://www.isasecure.org/en-US/Certification/IEC-62443-SDLA-Certification.

[29] ISA Security Compliance Institute, "SSA-311 System Security Assurance - Functional security assessment for systems, Version 1.82," 2014. [Online]. Available: http://www.isasecure.org/en-US/Certification/IEC-62443-SSA-Certification.

[30] ISA Security Compliance Institute, "ISASecure - IEC 62443-3-3 - SSA Certification," 2017. [Online]. Available: http://www.isasecure.org/en-US/Certification/IEC-62443-SSA-Certification.

[31] ISA Security Compliance Institute, "EDSA-312 Embedded Device Security Assurance – Security development artifacts for embedded devices Version 2.0," 2015. [Online]. Available: http://www.isasecure.org/en-US/Certification/IEC-62443-SDLA-Certification.

[32] ISA Security Compliance Institute, "SDLA-312 Security Development Lifecycle Assessment Version 3.0," 2014. [Online]. Available: http://www.isasecure.org/en-US/Certification/IEC-62443-SDLA-Certification.

[33] ISA Security Compliance Institute, "SSA-312 System Security Assurance – Security development artifacts for systems Version 1.01," 2014. [Online]. Available: http://www.isasecure.org/en-US/Certification/IEC-62443-SSA-Certification.

[34] ISA Security Compliance Institute, "EDSA-300 Embedded Device Security Assurance – ISASecure certification requirements Version 2.8," 2014. [Online]. Available: http://www.isasecure.org/en-US/Certification/IEC-62443-SDLA-Certification.

[35] ISA Security Compliance Institute, "SSA-300 System Security Assurance – ISASecure certification requirements Version 1.4," 2016. [Online]. Available: http://www.isasecure.org/en-US/Certification/IEC-62443-SSA-Certification.

[36] ISO, "ISO/IEC 27005:2011 Information technology – Security techniques – Information security risk management.," ISO, Geneva, 2011.

[37] J. Liedtke, "On μ-Kernel Construction," 1995. [Online]. Available: http://dilip.nijagal.com/technical/ukernel-construction.pdf.

[38] R. Kaiser and S. Wagner, "Evolution of the PikeOS Microkernel," 2007. [Online]. Available: http://ertos.nicta.com.au/publications/papers/Kuz_Petters_07.pdf.

[39] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch and S. Winwood, "seL4: Formal Verification of an OS Kernel," Jun 2010. [Online]. Available: http://ertos.nicta.com.au/publications/papers/Klein_EHACDEEKNSTW_10.pdf.

[40] S. Lescuyer, "ProvenCore: Towards a Verified Isolation Micro-Kernel," 20 Jan 2015. [Online]. Available: http://dx.doi.org/10.5281/zenodo.47990.

[41] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems,"

1975. [Online]. Available: http://web.mit.edu/Saltzer/www/publications/protection/, http://www.cs.virginia.edu/ evans/cs551/saltzer/.

[42] Information Assurance Directorate, “U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness. Version 1.03,” June 2007. [Online]. Available: https://web.archive.org/web/20110108022547/http://www.niap-ccevs.org/pp/pp_skpp_hr_v1.03.pdf.

[43] I. Furgel and V. Saftig, “Common Criteria Protection Profile “Multiple Independent Levels of Security: Operating System” [V2.03],” 31 Mar 2016. [Online]. Available: http://dx.doi.org/10.5281/zenodo.51582.

[44] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel and T. Holz, “kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels,” in *USENIX Security Symposium*, 2017.

[45] RTCA SC-205 / EUROCAE WG-71, “DO-178C: Software Considerations in Airborne Systems and Equipment Certification,” December 2011. [Online].

[46] CCMB, “Common Criteria for Information Technology Security Evaluation v3.1, Part 2: Security functional requirements,” 2017. [Online]. Available: https://www.commoncriteriaportal.org/files/ccfiles/CCPART2V3.1R5.pdf.

[47] A. Takanen, “Fuzzing For Software Security Testing & Quality Assurance,” in *EuroStar 2009*, Stockholmsmässan, Sweden, 2009.