



## D4.4

### Security testing framework

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

<b>Deliverable type:</b>	Report
<b>Deliverable reference number:</b>	DS-01-731456 / D4.4 / V2.0
<b>Work package contributing to the deliverable:</b>	WP4
<b>Due date:</b>	M28 – April 2019
<b>Actual submission date:</b>	13 <sup>th</sup> August, 2020

<b>Responsible organisation:</b>	UROS
<b>Editor:</b>	Thorsten Schulz
<b>Dissemination level:</b>	PU
<b>Revision:</b>	V2.0

<b>Abstract:</b>	Final security testing approach for the MILS platform and MILS platform components. This deliverable will contain a public report on developed security approach supplemented by confidential part about its application on SW components.
<b>Keywords:</b>	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 (ATSEC)

Alvaro Ortega (E&E)

Luise Müller (UROS)

Philipp Gorski, Holger Blasum (SYSGO)

We also thank EU reviewers for feedback on an earlier version.

## Disclaimer

The information in this document is provided “as is”, and no guarantee or warranty is given that the information is fit for any particular purpose. The content of this document reflects only the author’s view – the European Commission is not responsible for any use that may be made of the information it contains. The users use the information at their sole risk and liability.

## Executive Summary

This deliverable concerns the final security testing approach for the MILS platform and MILS platform components.

It describes how the testing approach matches high-assurance requirements from ATE for Common Criteria for Information Technology Security Evaluation (CC) EAL5+ (ISO/IEC 15408; Section 1.1) and IEC 62443 (Section 1.2). The testing approach leverages the results from WP1 on assurance techniques: The testing is derived from these requirements compliant to CC and IEC 62443. The tests are applied to the security attack surface using interface verification and fuzz-testing approaches.

Chapter 2 describes an implementation approach of concepts of defined in D4.1 for fuzz testing kernel drivers by a design that is partly in user-space, partly in kernel-space, thus crossing different address spaces and its capability to detect errors.

The tools introduced in deliverable D1.2 refer to commercially available products. Deliverable D4.1 describes robustness testing setups requiring specific adaptation. Therefore, the integration of the two categories into test frameworks for security tests is discussed in Chapter 3. It describes how the developed fuzz testing approach from Chapter 2 is integrated into the proven build- und test frameworks. Through the modular approach, this also applies integration of commercially available test applications. The exemplary use-case is on testing of separation kernel components and their interfaces using two different tools (adapted AFL, existing AbsInt rule-checker). The AFL approach was also adapted and applied to the TAS platform, using a comparable approach.

We conclude on lessons learnt using these methods to facilitate the security evaluation of the robustness and conformance of a kernel driver of a separation kernel.

# Contents

<b>Chapter 1</b>	<b>Introduction .....</b>	<b>1</b>
1.1	Requirements from Common Criteria on Security Testing .....	2
1.2	Requirements from IEC 62443 on Security Testing .....	4
1.3	Expectation Towards a Driver Testing Framework.....	4
<b>Chapter 2</b>	<b>Testing Approaches .....</b>	<b>6</b>
2.1	Static and Dynamic Analysis of Interface Usage.....	6
2.2	Exploration of Effects of Interface Payloads by Fuzz-Testing .....	6
2.2.1	Common Aspects .....	7
2.2.1.1	<i>Crash Recognition .....</i>	<i>7</i>
2.2.1.2	<i>Scalable Coverage Maps.....</i>	<i>8</i>
2.2.2	QEMU-Emulation Trace-Data Coverage-Feedback .....	8
2.2.3	HW-Tracing-Features Coverage-Feedback .....	8
2.2.3.1	<i>Adaptation to the separation kernel ecosystem.....</i>	<i>9</i>
2.2.3.2	<i>Performance on Sample Driver.....</i>	<i>9</i>
2.2.3.3	<i>Evaluation.....</i>	<i>11</i>
2.2.4	Source-Level Instrumentation Coverage-Feedback .....	11
2.2.4.1	<i>Adaptation for separation kernel ecosystem .....</i>	<i>12</i>
2.2.4.2	<i>Performance on Sample Driver.....</i>	<i>14</i>
2.2.4.3	<i>Evaluation.....</i>	<i>15</i>
<b>Chapter 3</b>	<b>Test Framework Integration .....</b>	<b>16</b>
3.1	Separation Kernel Test Framework Integration.....	16
3.1.1	Separation kernel project setup .....	16
3.1.2	General Test Framework Setup.....	17
3.1.3	Fuzzing Integration for the TF .....	17
3.1.4	External Tool Integration for the TF .....	20
3.2	Application TAS-Build-Chain.....	21
3.2.1	Structure and Components of the Testing Framework .....	21
3.2.2	Experimental Application .....	22
3.3	Results .....	24
<b>Chapter 4</b>	<b>Summary and Conclusion .....</b>	<b>25</b>
<b>Chapter 5</b>	<b>List of Abbreviations.....</b>	<b>26</b>
<b>Chapter 6</b>	<b>Bibliography .....</b>	<b>27</b>

## List of Figures

Figure 1 Security attack surfaces (see D4.1 [1]) .....	1
Figure 2 Separation kernel and kernel driver linking aspect and interface correlation.....	1
Figure 3 Separation kernel fusion-interfaces with the sandbox aspect .....	6
Figure 4 Classification of code coverage techniques.....	7
Figure 5 Dataflow in the KernelAFL approach (source: [8]) .....	9
Figure 6 KernelAFL findings graph.....	10
Figure 7 KernelAFL performance chart .....	11
Figure 8 GCC compilation stages (source: [9]).....	11
Figure 9 Source code instrumentation of basic blocks: left, source; middle, derived control flow tree; right, modified tree with locations of instrumentation injection .....	12
Figure 10 Instrumentation summary of GCC plugin applied on sample driver .....	12
Figure 11 Conceptual fuzz-test architecture with test agent .....	13
Figure 12 Fuzz-test architecture with test agent thread and corrected test pattern injection .....	13
Figure 13 Data flow in a fuzz-test of a kernel component .....	14
Figure 14 Separation kernel component compilation flow. ....	18
Figure 15 TF fuzzing integration project scheme.....	19
Figure 16 Information flow in automated server-based testing .....	20
Figure 17 Output of the robustness test run for the FiBuss protocol driver .....	23
Figure 18 Final stats of the robustness test run for the FiBuss protocol driver.....	24

## Chapter 1 Introduction

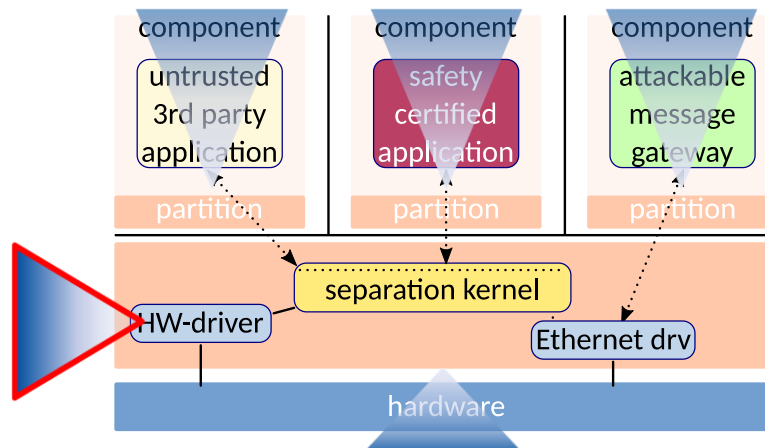


Figure 1 Security attack surfaces (see D4.1 [1])

A MILS system consists of several components, see for example Figure 1. These components may be provided by the separation kernel developer, the integrator or third parties. Previous certMILS deliverables presented the security concept for user-space components attached to the interfaces available in normal partitions.

For instance, a separation kernel product may specify the concept of kernel drivers to provide dedicated functional extensions to the kernel, which are not provided by the kernel via system calls or its Application Programming Interfaces (APIs) and cannot be implemented without running in privileged CPU mode as part of the kernel space. The most typical use case for this type of functional extensions is the necessity to support specific devices or peripherals of a hardware platform.

In such a design, the separation kernel specifies and provides the kernel driver framework implementing generic interfaces to permit such a functional extension. This interface specification defines the services exposed by the separation kernel to a kernel driver (kernel driver service API) as well as the callbacks to be provided by the kernel driver extension to the separation kernel (kernel driver callback API), see Figure 2 below.

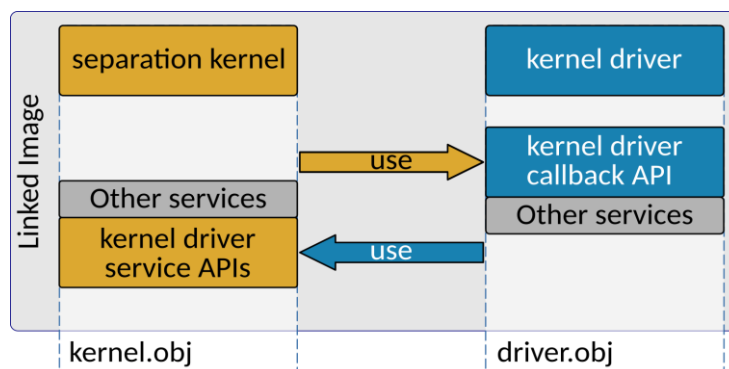


Figure 2 Separation kernel and kernel driver linking aspect and interface correlation

- The kernel driver service API defines the set of operations provided by the separation kernel that can be safely called from the context of a kernel driver under defined conditions.
- The kernel driver callback API defines the set of operations provided by the kernel driver that can be called by the separation kernel as well as other kernel driver objects. The separation

kernel passes calls from the user space down to these callbacks (Gate Callbacks) and triggers specific callbacks internally on the occurrence of certain events (Provider Callbacks e.g., initialization, health monitoring events, ...).

The kernel driver framework is part of the separation kernel product and responsible for loading and setting up registered kernel drivers. Therefore, each kernel driver to be added must be linked to the separation kernel object file within a defined address range and published to the separation kernel via dedicated symbols. Hence, for the development by the integrator or a third party, it is sufficient to provide the required configuration artefacts and the object file of a kernel driver as basic sources for the integration with the separation kernel via a project called “fusion project”.

Nevertheless, from the aspect of safety and security, there may be a lack of trust in such components. Especially, because they represent extensions to the separation kernel and its trusted software core. They may add additional security attack surfaces through the associated hardware ports (see Figure 1). Any weak implementation of the provided APIs or intended misuse of them has the potential to bring down the complete system with fatal consequences. Hence, ways for the evaluation of these parts and their robustness must be found. The testing must ensure the expected behaviour of the kernel driver in conformance to the specifications of the APIs.

To layout the testing requirements, security certification standards need to be obeyed. The security certification landscape is characterized by the generic Common Criteria (CC) and application-domain specific standards. For instance, it has been observed that the rigor of the CC suggests to focus on a small system/product or subsystem of a product [2]. The MILS separation kernel is such a product suitable for CC. In the domain of industrial automation and control systems (IACS), the standard IEC 62443 considers the security of entire plants and takes strongly into account the constant changes that need to be made to a plant, by putting great emphasis on the processes during the life cycle of an IACS. For instance, risk assessment is not just carried out at the beginning, but continuously repeated to achieve improvement. Railways and the track-side networks as distributed systems have a complexity comparable to IACS. The newly drafted CENELEC's prTS 50701 [3] is based primarily on the IEC 62443 standard in the field of cyber-security. Therefore the standard IEC 62443 is a clear choice for the environment of rail transportation. As the pilots are in the IACS domain (smart grid) and railway domain (railway and subway demonstrators), we chose IEC 62443 as certification standard.

In order to satisfy the requirements of these security standards, systems based on separation kernels, or the separation kernel itself, must pass the respective testing activities performed in an evaluation. In earlier certMILS deliverables, ISO 15408 (Common Criteria) and ISA/IEC 62443 were identified accordingly as two of the most relevant security standards applicable to separation kernels and MILS systems, where IEC 62443 is suited for larger systems built from many components, such as the pilots, and Common Criteria is suited for individual products, such as a MILS separation kernel. Therefore, separation kernel developers, integrators and operators take most benefit from testing compliant with standards like these. The subsequent two sections outline the testing approaches of the Common Criteria and IEC 62443. Additional constraints can exist in schemes that also define the evaluation methodology. Accordingly, these requirements may differ between the nations hosting certification bodies or the applicable certification scheme. These details are, therefore, not discussed.

## 1.1 Requirements from Common Criteria on Security Testing

ISO/IEC 15408 (Common Criteria, CC) knows three types of testing [4]. One of these testing efforts is performed by the developer of an IT-product and described by the assurance family ATE\_FUN during a CC evaluation. In the assurance family ATE\_IND, evaluators of an independent evaluation lab verify the developer testing. The remaining type is a penetration testing effort (pen-testing) performed as part of the vulnerability assessment within the AVA class, also by the CC evaluation lab. The required degree of rigor in the assessment of all aspects is dictated by the Evaluation Assurance Level (EAL) claimed in the Security Target (ST), which, as well, specifies the security

features of the product. The EAL also determines the degree of detail required for the product's documentation, as it needs to provide all necessary information to the evaluator. For the purposes of the present document, an assurance level of roughly EAL5 (EAL5+) is assumed.

Like the independent testing, the vulnerability assessment is planned and executed by a CC evaluation lab. Therefore, it largely depends on the judgment of the evaluator and, possibly, requirements of the national certification scheme. The evaluator takes into account all knowledge of the lab collected during the evaluation of other aspects. The pen-testing can, therefore, differ depending on the chosen evaluation lab and not be planned by the developer of the product.

The independent testing of ATE\_IND is also defined by the evaluation lab, but often influenced by the developer's own test approach. While the evaluator will focus on gaps or weak points of the existing testing, these activities will often build on an existing testing framework.

Since the AVA class does not impose any requirements on the developer's testing framework, this section seeks to describe the implications of the families of the ATE assurance class. As ISO 15408 is a flexible standard applicable for all kinds of IT-products, its requirements are generic. For example, the standard does not tell the developer to test the product in a specific way, using certain methods, or even dictate tools to be used. Its requirements rather concern the developer's documentation describing his testing approach and recording the test results. This information must enable the ATE evaluator to determine whether the testing approach is sound. Furthermore, the developer needs to demonstrate that the testing relates to the security functions that are claimed and modelled in the ST.

In general, the CC testing is performed by exercising interfaces. Those can be external interfaces (TSFIs) or internal interfaces between the subsystems and modules of the test target. To show the correspondence between the testing according to the test documentation and the TSFIs, the developer provides an analysis relating the two. This tracing allows the evaluator to determine the extent of test coverage obtained by the developer testing and to identify possible gaps (ATE\_COV). At EAL5, all TSFIs need to be tested. Similarly, the developer provides an analysis that the testing exercises the internal interfaces (ATE\_DPT).

The CC define TSFIs as all kinds of interfaces between the portion of the test target that comprises the security functionality (TSF) and other parts or the environment, provided they have a relationship with that security functionality. Therefore, all of those interfaces are in principle subject to testing efforts. In practice, direct testing of some types of interfaces can be more difficult than that of others. For instance, it may be more difficult to test interfaces between hardware and software as this may require instrumentation. Likewise, the testing of internal interfaces in ATE\_DPT can be achieved either indirectly, exercising the TSFIs, or by testing portions of the test target separately, e.g. within a test harness.

Some relevant detailed testing issues are left open by the standard and are in practice handled by the experienced CC evaluator or additional requirements issued by the certification body. For example, although, the CC do not use the term 'attack surface' and treat all TSFIs evenly, it may make sense to intensify the testing activities for interfaces that are directly accessible by an attacker. The CC also do not require the qualification of test tools and test frameworks. Furthermore, the standard does not specify with what intensity a TSFIs need to be tested (e.g., how many different input parameters need to be tried). Thereby, it does not strongly influence the design of a test framework implementing, for instance, fuzzing techniques.

In summary, the CC make it rather easy for the developer to achieve compliance with its testing related requirements as it accepts all kinds of approaches and frameworks, as long as it has the sketched properties. At high EAL, the testing needs to cover all TSFIs as well as internal interfaces between the modules. Correspondingly, the supplied documentation needs to achieve the resolution required for the evaluator to assess whether this is the case. For any chosen test approach, the developer needs to provide test plans, different tracings or mappings to other parts of the documentation, and sensible records of the test results (actual outcomes). The developer can benefit from a test framework that produces some of this documentation automatically, however that is not required for the compliance with the CC.



## 1.2 Requirements from IEC 62443 on Security Testing

In IEC 62443 normative text, the testing needs are identified and included at several levels for each of the parts within the scope. However, the focus of these testing approaches is mostly on functional testing, integration testing and patch verification testing.

Specific security testing, is covered by part 4-1 [5] at the level of IACS components. It describes the testing methods in the table below.

Name	Description
Security requirements testing	This testing focuses on verifying that all the security requirements in the security requirements specification (SecRS) have been met. Functional, negative, boundary, performance and other types of standard testing are performed on the security capabilities in the SecRS.
Threat mitigation testing	This testing is based on threat trees created from the threats identified in the threat model and ensures that the mitigations designed and implemented in the product are effective in stopping the given threat. Testers design their tests to attempt to thwart the mitigation using the type of threat identified.
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 of the implementation or its defence in depth design.
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 approach is unstructured and depends on the skills and knowledge of the attacker. The tester tries to play the role of an attacker. It 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 tools, techniques and methods to be used during the security testing are left open to testers, but fuzz-testing is a recommended practice. Detailed requirements regarding the testing may arise from a given ISO/IEC 62443 certification scheme. At the time of writing of this document such schemes are still in an early stage of evolution.

## 1.3 Expectation Towards a Driver Testing Framework

The security architecture, asserts strong assurance for the interfaces of a MILS system between the application components and the separation kernel. An application component is not allowed to have any other interfaces (beyond higher-level abstractions of those). In general, a separation kernel does, however, expose further interfaces to other components of an integrated system.

The introduction discussed the additional need for privileged driver components to adapt the separation kernel to specific hardware configurations, or to introduce additional interfaces to application or system specific hardware. When, for technical reasons, these driver components are

realized as a kernel device driver, the internal interfaces cannot provide the same security strength as the external kernel API, due to the same address space. Therefore, organizational measures should be provided, such as a testing framework. Testing of internal interfaces, as covered by the testing framework, is required for high assurance levels (EAL5+) according to CC.

Kernel drivers are often provided by Integrators. Hence, the testing framework must be accessible and usable by developers of the separation kernel, integrators and third parties.

- A third party developer and the integrator of a MILS system typically have minor experience in and resources for testing of added kernel components compared to the SK provider.
- An integrator, operator or evaluation facility may require the demonstration of testing activities within own premises due to quality assurance reasons requiring independent tests.
- An integrator has no access to the source code of the SK. This limits the tests to at most partial instrumentation.
- If the SK provider runs the driver testing framework, it may not have access to the source of the driver or a third party component.
- The testing approach must support all target architectures of the SK.
- The result of any test run must be reproducible. The payload data of a fuzz-test leading to a crash must be stored. Each test run must not depend on prior payload data to be reproducible.
- Testing of internal interfaces may be needed to verify a defence-in-depth approach.

## Chapter 2 Testing Approaches

The following sections describe the tool set implemented to test kernel extensions through their kernel APIs by the collaboration. For other security-testing aspects, refer to deliverables D4.1 [1] and D1.2 [6].

This kernel testing approach consists of analysing the usage of the internal interfaces followed by their fuzz-testing using data payloads sampling the parameters.

### 2.1 Static and Dynamic Analysis of Interface Usage

The basis for the proposed method to evaluate the behaviour of the untrusted kernel driver is the observation of its operation at runtime in the intended production environment. The observability should be ensured independent of whether the source code is available or only an object file of the kernel driver without its sources.

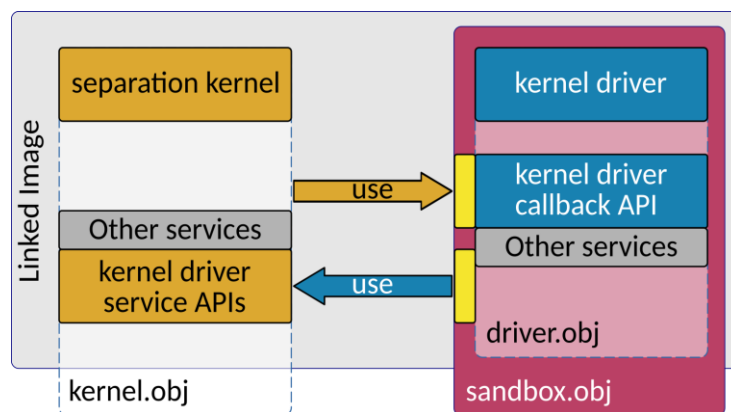


Figure 3 Separation kernel fusion-interfaces with the sandbox aspect

Hence, means for checking and manipulating the object file must be found to run the kernel driver in a sandboxed environment allowing to analyse the interactions with the separation kernel, as well as other kernel space entities during full tests of the complete kernel driver object file.

As entry point for the sandbox-approach, a second kernel driver (the sandbox) is prepared. The specific methods and modifications are detailed in a confidential supplement. During the separation kernel fusion, this additional sandbox kernel driver is linked with the kernel driver under investigation.

Furthermore, the sandboxing is achievable with the tools provided as part of the separation kernel ecosystem to avoid any external tool dependencies.

### 2.2 Exploration of Effects of Interface Payloads by Fuzz-Testing

Fuzz-testing is a testing methodology that executes the target with mutated data payloads to potentially trigger faulty behaviour. When data payloads are mutated randomly, it takes long to achieve good code coverage of the target. Better performance is achieved through code coverage feedback. A target's code coverage (classification in Figure 4) can be traced with help of hardware profiling features, such as Intel PT [7] or ARM CoreSight [8]. The profiling subsystem of a system emulator, such as QEMU, is also considered a "hardware" feature, as it does not modify the test target.

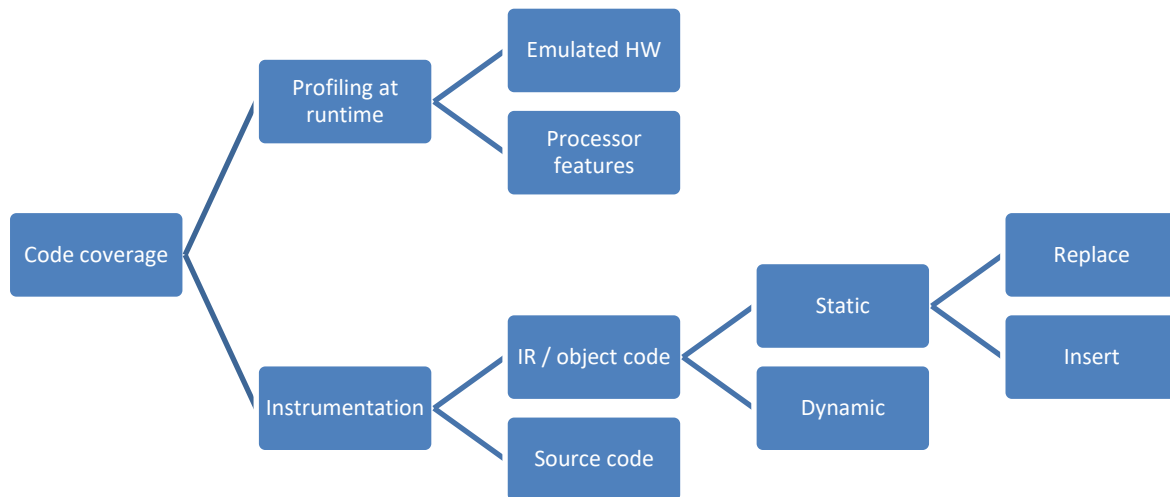


Figure 4 Classification of code coverage techniques

In contrast, without HW-based profiling support, source- and object code instrumentation can also be used to analyse the code coverage. Additionally, binary instrumentation can be applied dynamically at run-time. However, this technique is more complicated compared to static and source instrumentation, especially in the targeted kernel-space. Section 2.2.4, discusses the source instrumentation technique, as it is most versatile, scales well and produces only little overhead and adds only minor technical requirements. It is the preferred approach, as owners of the source of a specific test-target kernel driver (only the test-target needs to be instrumented) know best to fix the identified issues, while keeping the test framework as simple as possible. The source instrumentation technique will trace on branch level with a coverage map compatible to the AFL fuzzer [9].

Fuzz-testing is often understood to be a generic test methodology. Nonetheless, it still requires a test harness to input the test payload into the test target at the location of interest. For example, the AFL fuzz-testing tool, by default, starts the test candidate and injects the test data payload via the process's standard input channel. Components of a MILS system have different interfaces that must be tested. As discussed in Chapter 1, extensions of the kernel have APIs, which are not exposed externally. Furthermore, if a fault occurs (e.g., a crash) in the kernel, the system will be unable to document the incident. As a result, the fuzz-test infrastructure must reside outside of the target system: either on a separate machine or outside of the virtualized environment. The additional communication paths affect the performance negatively.

The large number of calls of the kernel driver API and further component specific calls are another impediment to a simple, generic framework.

Another shortcoming of generic test cases executed by fuzz-testing are the limited options for the verification of results. Typical generic fuzz-testing infrastructures can only capture rogue misbehaviour that leads to a crash of the target. Non-faulting dysfunction or information leakage cannot be detected without further custom analysis. Fuzz-testing is more of a robustness testing technique than a catch-all approach for security testing.

## 2.2.1 Common Aspects

### 2.2.1.1 Crash Recognition

For efficient test execution, the logic of the fuzzer needs to detect the crash of a target as early as possible since lock-up situations (timeouts from the viewpoint of the external fuzzer) are very costly performance-wise. Typically, faults trigger non-maskable interrupts (NMI) to give the OS a last chance to react or at least log the fault. These NMI are caught by a supervision subsystem of the OS kernel.

In a separation kernel, this can be made the task of the Health-Monitor subsystem (HM). The HM provides hooks (function tables) for a number of events, amongst others, for a kernel panic. We use

this callback to notify the fuzzer and to save further information related to a crash. To notify the external fuzzer from within QEMU, there are two ways:

- 1) Trigger a HW-port-based virtual QEMU device (“pvpanic”) that notifies via the monitor channel.
- 2) When address tracing is used, setup a trigger that fires upon execution of the mentioned callback function.

Using this approach, most faults can be caught. However, soft faults that do not trigger immediate reactions or exceptions are harder, if not impossible, to find in that way. Since the fuzzer does not know the expected behaviour, this aspect requires a fundamentally different approach.

### **2.2.1.2 Scalable Coverage Maps**

General-purpose fuzz-testing tools, such as AFL, have to cope with applications of different sizes. Coverage maps tracing control-flow on branch-level, using an address trace or address-space map, can become very large and inefficient to process (see next section).

AFL instead implements a compressed map of a fixed size (typically 64 kB or less). The location on the map is computed from the lower bits of the x-or of the current and the previous branch address. This algorithm has proven to perform well, however, there is not necessarily a unique mapping from the coverage map to source locations. Such a relation is not required for fuzz-testing. The bare information of a change in the map in new locations triggers the fuzzer’s interest to fine-tune the payload to explore this newfound path.

### **2.2.2 QEMU-Emulation Trace-Data Coverage-Feedback**

When QEMU is running in emulation mode, i.e. with just-in-time machine-code translation, QEMU can be configured to output detailed information about its execution. However, this information can be too detailed, as it goes beyond address contents and time-stamps of the program counter (instruction pointer). A single boot process of an operating system may produce several gigabytes of trace-data.

As a result, this feature is limited to non-time-critical applications, e.g., tracing of functional test cases. Fuzz-testing requires several hundreds and thousands executions per second, thus trace-data collection is a performance bottleneck.

### **2.2.3 HW-Tracing-Features Coverage-Feedback**

Schumilo et. Al. [10] demonstrate “coverage-guided kernel fuzzing in an OS-independent and hardware-assisted way by utilizing QEMU and Intel’s Processor Trace (PT) technology”. The sources for the KernelAFL (kAFL) framework were also made available to adapt and apply to other targets.

Their approach, relies on the Intel PT execution address-tracing unit to record the control flow of a target. Like the former approach, trace data can quickly accumulate. However, PT can already filter certain events or instruction types, e.g. Change of [control] Flow Instructions. The modified QEMU version QEMU-PT has extra capabilities to filter the data stream from PT and produce coverage maps that can be evaluated by the fuzz-tester kAFL.

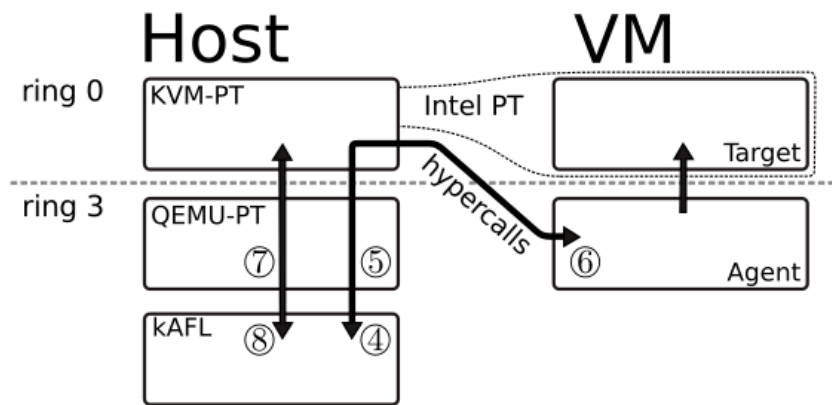


Figure 5 Dataflow in the KernelAFL approach (source: [10])

After initialization, the repetitive control flow in Figure 5 starts by generating the test data pattern in kAFL (4). This data is passed to QEMU-PT (5), which forwards it to the test-execution agent and activates PT trace sampling. The agent then executes the target in (6) and afterwards acknowledges successful execution back to QEMU-PT. QEMU-PT then requests the trace data from KVM-PT (7) and translates it to the coverage map for evaluation by kAFL (8). The findings from the changes in the coverage map can then be used to fine-tune the next test-data pattern.

### 2.2.3.1 Adaptation to the separation kernel ecosystem

The kAFL approach requires multiple tool adaptations. First, the host system's operating system kernel needs special support for the Intel Processor Trace (PT) functionality within the Kernel Virtualization Module (KVM). The patch by the kAFL authors measures ~50kB in size. It is available for the Linux Kernel in version 4.6.2. We have updated the patch to work with Linux 4.9, as it is planned to be a long-term-support kernel (Jan, 2023<sup>1</sup>). The effort is feasible, but must be accounted for. There is no support for other host OS.

QEMU is the emulation and virtualization application used as a tool for development and test of separation kernel systems. The separation kernel that we analysed is shipped with QEMU 2.7.1, released 2017. KernelAFL provides a patch for QEMU 2.9.0 to integrate support for a dedicated SHM device, capturing of hypercalls and accessing, filtering and controlling PT. We have updated this patch to the closest available QEMU version within the Debian Linux distribution (version 2.10.1). The effort to update the QEMU patch, despite the small version number difference, was much higher compared to the Linux patch. Further updates to newer versions are considered a heavy impediment.

Lastly, the kAFL fuzzing application is a one-off publication with neither long-term support nor maintenance community. It is, however, open-source.

### 2.2.3.2 Performance on Sample Driver

The authors packaged samples for kAFL on Linux and Windows. The Linux sample, which contained an intentional bug, was translated to the form of a separation kernel driver as an example of a SW component. Two "magic" data payloads lead to a crash are different compared to the ones used for source instrumentation in section 2.2.4.2. KernelAFL provides dot-graph based output of the fuzzing run sketched in Figure 6.

<sup>1</sup> <https://www.kernel.org/category/releases.html>

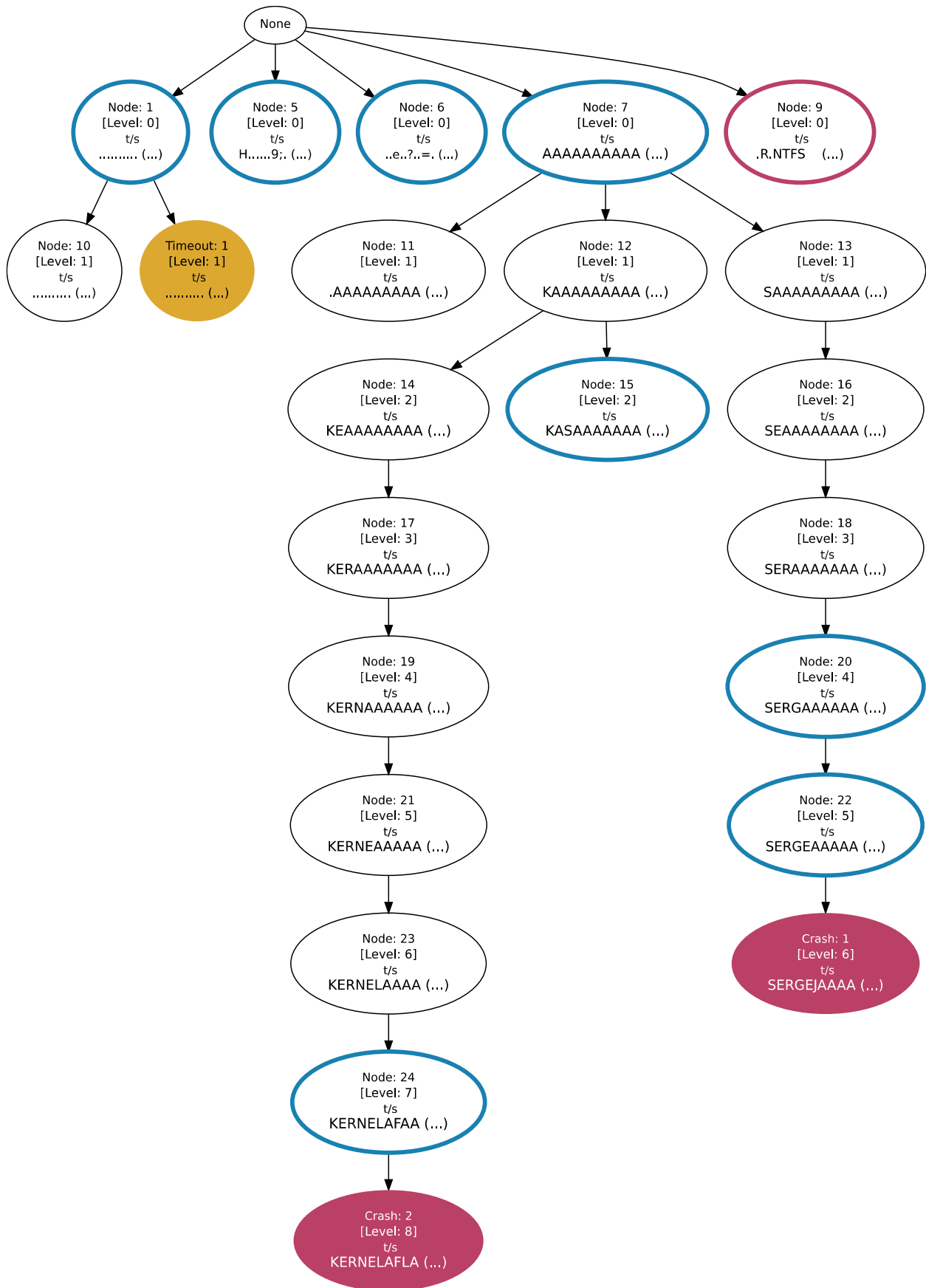


Figure 6 KernelAFL findings graph



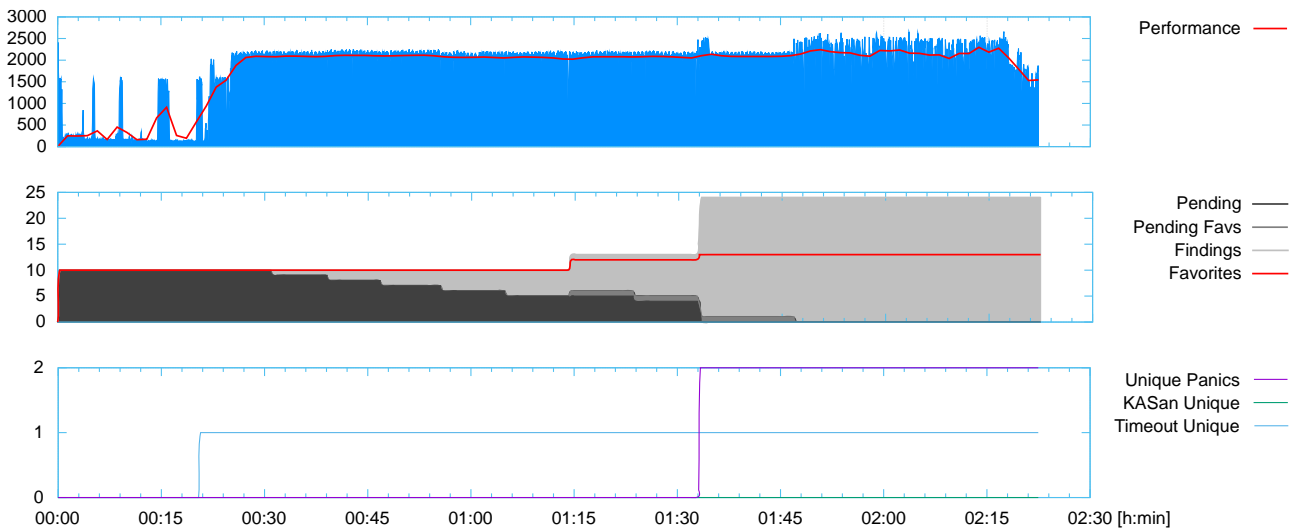


Figure 7 KernelAFL performance chart

The first chart in Figure 7 shows the executions per second, which most the time averages around  $2000\text{ s}^{-1}$ . Below, in the middle of Figure 7, the chart displays metrics of the fuzzer based on the feedback from coverage and the success of specific paths. The bottom chart shows crashes of the target. Co-incidentally, both crashing paths were found within a few minutes apart. The detected timeout is a false positive due to an unrelated high-priority job on the test machine that confused the power-saving CPU governor. This may also account to the changing performance in the first minutes of the fuzz-job.

### 2.2.3.3 Evaluation

The kAFL performance measured in test executions per second is highest of all approaches trialed – roughly four times better than the source-level instrumentation approach described below.

However, the necessary adaptations to host-tooling discussed in section 2.2.3.1 are quite severe. They are deemed problematic for long-term support and easy distribution. Furthermore, this approach with Intel PT is limited to Intel CPUs. The equivalent technology for ARM architecture processors, CoreSight, was not covered due to time constraints but should be followed for future work.

## 2.2.4 Source-Level Instrumentation Coverage-Feedback

The generation of a separation kernel system's boot image consists of many steps. First, all components are compiled to object code. These fragments are then linked together to the binary and finally merged into the boot image.

Compilation and linking may happen at different times. A system integrator, who wishes to add his component to the kernel image typically obtains the operating system library from the OS component supplier as a binary object. The new component that is the target for the tests, however, is available as source code. The instrumentation is only required on the test-target's source, not on the whole kernel. Instrumentation is applied in the compilation process of the component.

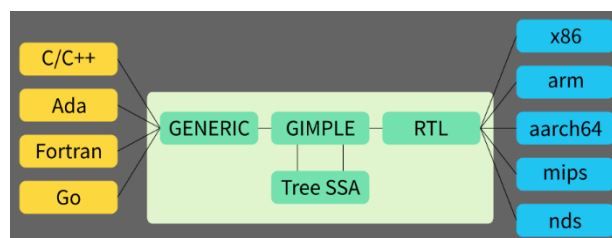


Figure 8 GCC compilation stages (source: [11])

The compilation process of GCC undergoes several intermediate stages and code representations (Figure 8). Without going into detail, each stage can be extended with custom plugin modules. The



program control flow, which is best suited for coverage analysis, is found in the Gimple intermediate language stage. The control flow is composed of basic blocks with linear execution, as seen in Figure 9. Our plugin function hooks are called in this stage to insert the source instrumentation for code coverage recording. At the beginning of each block, a function call is inserted that takes the current instruction address, x-ors it with the former address and uses the result to index into the coverage map to increase the counter at that index.

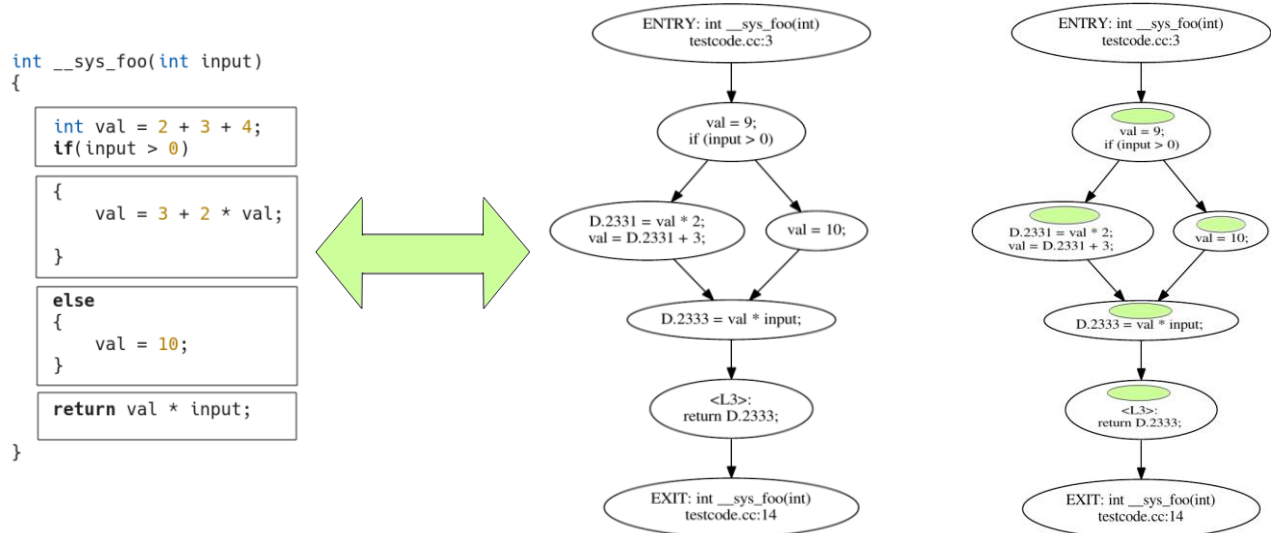
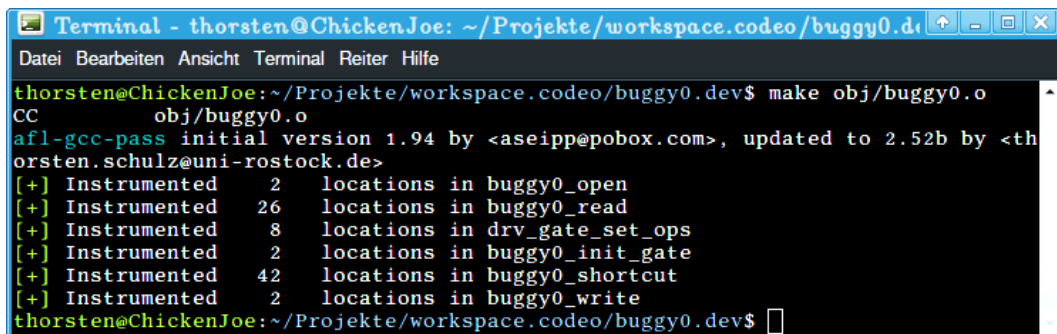


Figure 9 Source code instrumentation of basic blocks: left, source; middle, derived control flow tree; right, modified tree with locations of instrumentation injection

The instrumented test target object (Figure 10) is then linked with the kernel. The whole approach is independent of the underlying hardware architecture. As above, this instrumented object needs assistance by special test agents, utilities that transfer the data and trigger the execution, which must also be linked. The data and execution flow is explained in the next section.



```

thorsten@ChickenJoe: ~/Projekte/workspace.codeo/buggy0.dev$ make obj/buggy0.o
CC      obj/buggy0.o
afl-gcc-pass initial version 1.94 by <aseipp@pobox.com>, updated to 2.52b by <thorsten.schulz@uni-rostock.de>
[+] Instrumented 2 locations in buggy0_open
[+] Instrumented 26 locations in buggy0_read
[+] Instrumented 8 locations in drv_gate_set_ops
[+] Instrumented 2 locations in buggy0_init_gate
[+] Instrumented 42 locations in buggy0_shortcut
[+] Instrumented 2 locations in buggy0_write
thorsten@ChickenJoe: ~/Projekte/workspace.codeo/buggy0.dev$

```

Figure 10 Instrumentation summary of GCC plugin applied on sample driver

### 2.2.4.1 Adaptation for separation kernel ecosystem

By principle, testing should be done as early as possible. As such, most of the testing of embedded systems is executed in virtual environments on a development or dedicated testing host. The system emulation and virtualization software QEMU is a prominent application that is also used as a testing environment for the separation kernel.

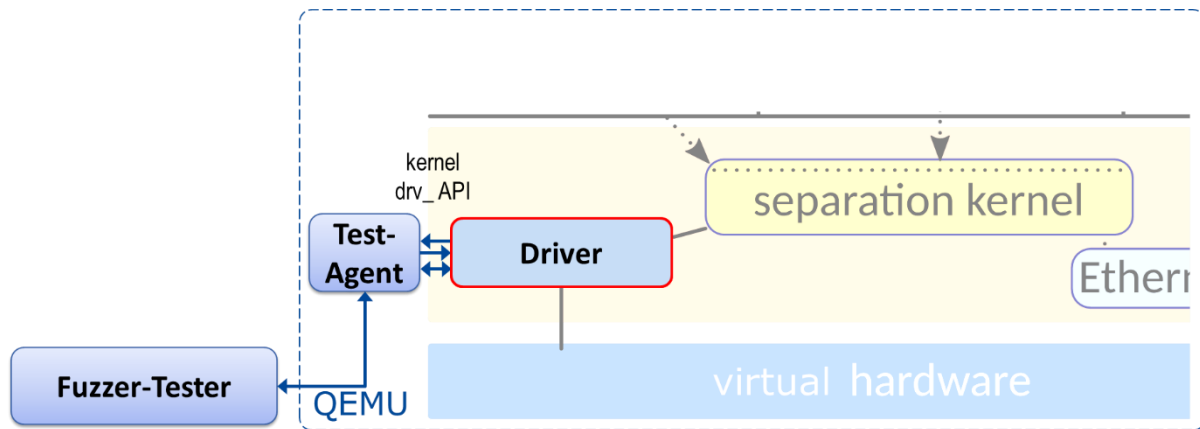


Figure 11 Conceptual fuzz-test architecture with test agent

As discussed in the previous section, the test database has to reside outside of the test target environment (see concept Figure 11, based on system architecture in Figure 1). We focus on QEMU system virtualization. However, the same concept also applies for a setup of separate test executor and test target devices connected by a network or debugging link.

The next feature in Figure 11 is the test agent. It receives the test patterns from the external test database and executes it onto the target component via the desired interface. The test agent either implements different APIs or is a specific replaceable component. However, the architecture in Figure 11 is technically not feasible.

First, it is problematic to “magically” induce data into the test-target’s memory without knowing all necessary side-effects. There needs to be some sort of device to represent an entry gate for data from the environment. Performance wise, a PCI-shared memory segment would be best. In the version of the separation kernel we used, the API for PCI-device drivers can only be used in the system service or user-space context, not in the kernel directly. Fixed SHM segments are not supported by QEMU on a modular or configurable basis without source modifications.

Secondly, in order to use user-space thread management, each call to a driver must be initiated by a user-space application at some point. There are exceptions to this rule, such as driver initialization and interrupts by hardware, but these cannot cover enough code of the tested component.

For these reasons, we split the test-agent into two parts: one as an extra partition in the user-space and one agent as a driver in the kernel-space, see Figure 12. In this setup, the user-space agent talks to the fuzzer via a TCP-based-link that is included in the separation kernel development framework, and is able to multiplex several TCP connections (the “multiplexer”). This concept also works in non-QEMU-setups.

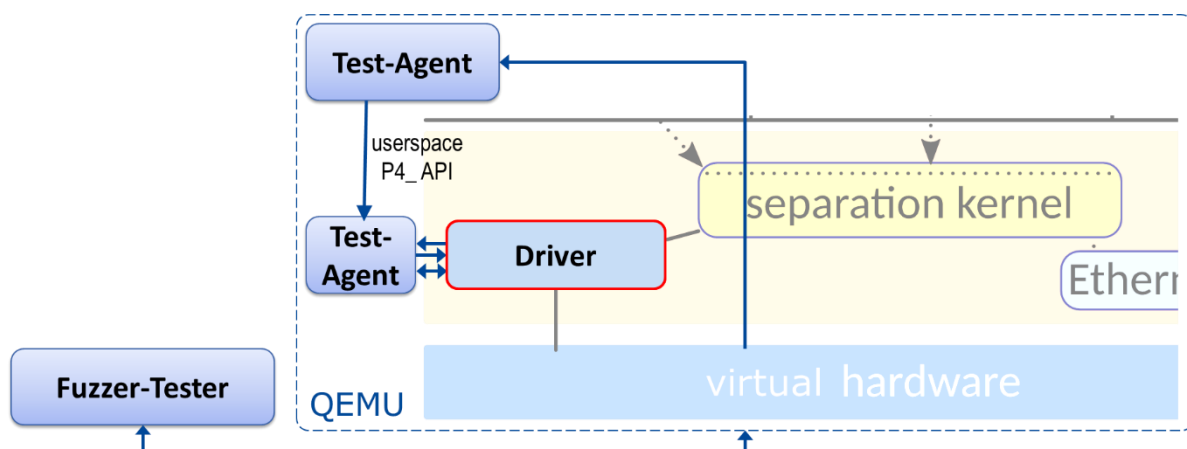


Figure 12 Fuzz-test architecture with test agent thread and corrected test pattern injection

The whole process for a single test run is shown in Figure 13. At first and once (not shown), the internal network of the host is configured and the separation kernel multiplexer host-service is

started. The target test-image is generated with the instrumented components and the test-agents linked in. With this, QEMU boots, however waiting for the fuzzer to connect for control.

When the environment is set up, the test starts by executing AFL. AFL is passed the remote communication proxy (RCP) as a pseudo target-binary, which emulates the remote target by forwarding the test-data payloads and “committing suicide” if the target machine faulted. With this architecture, AFL and QEMU can be left unmodified.

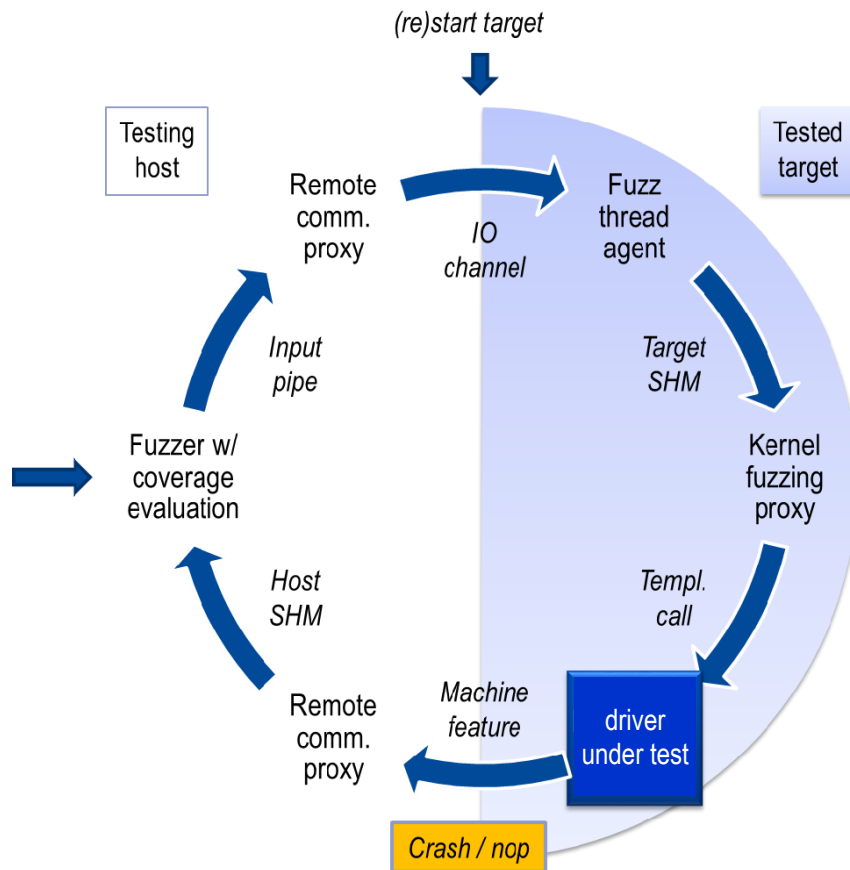


Figure 13 Data flow in a fuzz-test of a kernel component

At initialization, RCP runs the target to the start-point, when the test agent signals its Hello-message. RCP triggers QEMU to take a snapshot of the virtual machine for the following invocations. Now RCP reads the new test data payload from AFL through the standard input stream and forwards it via the multiplexer to the test thread agent. The thread agent invokes the kernel driver fuzz agent driver via a control command and a separation kernel SHM region. Finally, the target driver component is called by the fuzz proxy agent. If the test run succeeds without faults, latter returns the coverage map and signals the finalization back to the RCP.

In case of the QEMU environment, the return is optimized. The fuzz proxy fires a hardware trigger that stops the virtual machine and notifies the RCP via the control channel. The RCP can then do a memory dump of the coverage map in the test target, which proceeds much faster than the IO-channel path. This mechanism also works in a fault situation, where the multiplexer-return-channel would not be available anymore. The RCP copies the dump of the coverage map to AFL’s coverage map SHM and signals either a crash or readiness for the next test run.

#### 2.2.4.2 Performance on Sample Driver

The described architecture has proven effective, but there is still potential for optimization. The separation kernel debug-channel-I/O multiplexer has been identified as a major performance bottleneck. It also requires a developer license, which limits fuzzing on a larger number of machines or in a data center (see parallel fuzzing in [9]).

For future refactoring, the multiplexer should be replaced by a simpler data channel. For example, switching the return data (~64kB) from a multiplexer transmission to the QEMU memory dump has reduced per test execution time by 9 ms, in this case, increasing executions / second by factor 10.

The architecture was trialled on a low-performance development host machine. It achieved about 500 to 1000 test-executions per second. A buggy driver was equipped with three hidden “code bananas”. It contains an invalid pointer access, an infinite recursion and a division by zero. One of the three was typically found within minutes (5...100 min). However, to detect all three would take hours.

#### **2.2.4.3 Evaluation**

The sample driver trialled in the previous section was only tested on one API call. The amount of work required to set up a database of templates for a larger set of major API calls was underestimated and is still work in progress.

For simplification of the security-testing framework, this approach and the sandbox methodology in Section 2.1 should be merged. This was not planned for initially, however scanning and identification of API calls must be automated and would be a redundant effort otherwise.

## Chapter 3 Test Framework Integration

The pilots within the certMILS project are based on different Build-Tool-Chains and build approaches, which are referred to as the frameworks. System and component development and lifecycle maintenance follows processes with many actions, including testing. Where feasible, these action items are automated within these frameworks. New items are typically added to “hooks”, which are executed at defined locations of the process. Each hook iterates through its (extensible) list of actions. These action hooks of frameworks are often clustered into sub-frameworks. In this case, we will hook into the test-framework’s actions.

Unfortunately, security testing is quite application specific due to the individual attack surface exposed by each test target, so automation of security testing tools is limited. Another problem is, that security testing is not transparent to refinement (as was discussed in other project deliverables), i.e., there is only limited value to testing an isolated component in a reduced simulated environment, compared to running a security test in the assembled system – which is much harder to accomplish and automate.

In this chapter we show how we integrate the developed fuzz-testing tools and the code analysis tool AbsInt rule-checker (for static security code checks) within the separation kernel test framework. For integration within the TAS-Platform framework we reached a proof of feasibility status.

Other security testing tools mentioned in D4.1 and D1.2 are partially already integrated as an optional framework extension, such as XML checker (analysis of project configuration), linter, API checker, code analyser, etc.

### 3.1 Separation Kernel Test Framework Integration

#### 3.1.1 Separation kernel project setup

For a better understanding of a typical separation kernel ecosystem, some relevant basic project types are introduced upfront to the basic operations of the separation kernel test framework (TF).

- **Application Project**  
An application project is a specific project type to develop, configure, and build application artefacts for the separation kernel.
- **Kernel Driver Project**  
A kernel driver project is a specific project type to develop, configure, and build kernel driver artefacts for the separation kernel.
- **Kernel Fusion Project**  
A kernel fusion project is a specific project type to merge kernel-related project artefacts (e.g., kernel drivers) with the binary and configuration of the separation kernel.
- **Integration Project**  
An integration project is a global configuration artefact for a separation kernel image that will be built to run on a hardware target. In its static configuration it contains all resource setups, assignments, communication paths, access rights, or configuration limits. The configuration refers to the assigned project artefacts:
  - Application projects for resource partitions and processes.
  - Kernel driver projects.
  - Kernel fusion projects.
  - ...

Building an integration project means that all referred artefacts must exist, their binaries are merged into a final boot image, their configuration artefacts are compiled and merged into a final boot image.

### 3.1.2 General Test Framework Setup

The test framework for separation kernel (TF) organizes its tests via different abstraction layers as follows:

- Test suite

A test suite represents the highest level of abstraction. It is a collection of test cases, test sets, documentation, and global configuration settings. As criteria for the association with a certain test suite, a requirement document or certain API is most commonly applied. Running a test suite implies that all associated test cases are processed within their assigned test set configurations and a final reporting on the passed or failed test cases is generated (with the related documentation artefacts).

Furthermore, a test suite can contain further tools that can be used by the test cases (e.g., dedicated libraries) or for the result evaluation (e.g., special purpose scripts).

A test suite contains at least a single test case with the related test set.

- Test case

A test case represents the specific content for the application project that is running as test secondary on the test hardware. This includes the source code, test vector inputs, Makefiles, and documentation artefacts. Each test case is assigned to a dedicated test set.

Furthermore, a test case can provide dedicated extensions and customizations if it is needed for its specific test purpose (e.g., kernel drivers, make targets).

Beneath the binaries of the test case itself, the test cases are developed against a library of the TF to communicate the status and results to a controller counterpart (test primary), when running on a hardware target.

- Test set

A test set represents a specific configuration for the basic environment a test case is using and possibly customizing. This includes the basic separation kernel projects (application, integration ...), library or tool paths, modification snippets, and documentation artefacts.

In general, all testing performed by the TF is applied upon the project ecosystem of the separation kernel itself. Dedicated tooling or build steps are controlled via path variables and customized Makefiles. With this flexible solution, test flows can be customized, extended, or adjusted individually for each test case.

The test flow of the TF is suited to automatically build, run, and evaluate test suites with few make targets in a defined environment (e.g., chroot) that provided the needed toolchain.

The test suite run can be the compilation itself, with later evaluation of the build artefacts, or a complete execution of the test cases on a hardware target. For those runs on real or emulated hardware, a primary/secondary control and communication scheme is applied. The primary runs locally, or on a server that performs the automatic testing, and connects to the test case secondary on the application via the chosen communication peripherals (e.g., serial, Ethernet). Both parts of this communication chain are customizable.

### 3.1.3 Fuzzing Integration for the TF

The integration of the security fuzzing extension is applied as a derived test case and test set.

Assuming a full set of separation kernel projects for the targeted scenario already exists and is runnable, the specific components that shall be targeted by the fuzzing, need to be re-compiled with a modified compiler flow as depicted in Figure 14, to support the required instrumentation plugin for GCC.

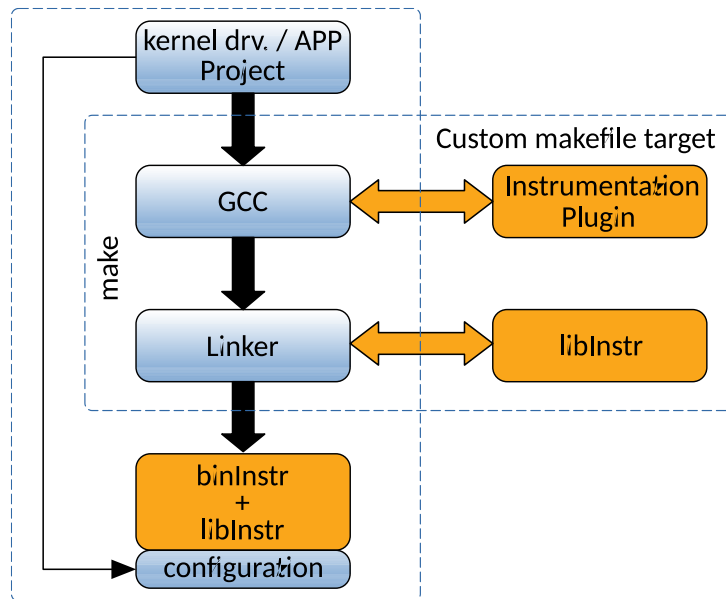


Figure 14 Separation kernel component compilation flow.

In the following description, it is assumed that mainly application or kernel driver projects are subject of the fuzzing.

Furthermore, the following tools are required:

- A predefined instrumentation library (libInstr), which provides the function call and data symbols the instrumented object code can be linked against. It implements the operations to retrieve the trace information and pass possible inputs to the fuzzer instrumentation points. It can be a customizable component with sources or a predefined one as static library binary.
- A configurable kernel driver component, which acts as handler for the gathered trace and input data. For the instrumented applications APP (binInstr) or kernel driver (binInstr), this kernel driver component is invoked by the inserted libInstr functions to write trace data or read input data. It can be a customizable component with sources or an existing one as pre-compiled binary.
- A customized test supervisor host, which operates as controller and processes the data exchange from the host side with the kernel driver component at the target side. It can be a customizable component with sources or an existing one as pre-compiled binary.
- Makefile adjustments, which serve the purpose of adding the new fuzzing targets as well as pre- and post-processing commands that trigger the right tools.
- Optional tools such as scripts for the pre- or post-processing.

Under consideration of an existing and runnable project set, an automated fuzzer flow for the TF is as follows:

1. The desired application and kernel driver projects subjected for the fuzzing are marked in the project configuration via a selectable option.
2. The separation kernel project configurator or a dedicated script clones the existing project set to generate a new combination of test cases and test set as export. Thereby, the application and kernel driver projects with the selected fuzzer option are now configured to be compiled with instrumentation. The fusion project integrates the kernel driver component (binInstr, referred to as “Test Agent” or “Kernel Fuzzing Proxy” in the run-time flow of Figure 12 and Figure 13) and the extracted configuration from the instrumented projects. The integration project integrates the user-space part of the fuzzer (“Test Agent” in Figure 12;



- “Fuzz Thread Agent” in Figure 13). These extensions are referenced by the corresponding entries in the test set.
3. The new test cases and test set derived from the separation kernel project set can be added to an existing test suite with focus on security / robustness related testing, or a new one is set up for this purpose.
  4. A generic or customized test primary is added to provide the controller part of the fuzzing from the host side (e.g., referred to as “Remote Communication Proxy” in Figure 13 as well as the AFL executable)
  5. Finally, the make targets for the possible post-processing and evaluation of the result logs can be added. (As crash results from fuzzing need manual post-processing to filter false positives and common-cause reduction, this currently results in a failed test case and a related notification.)

In consequence, the final derived combination of test cases and test set produces a boot-image as depicted schematically in Figure 15. The approach is that each instrumented component is targeted by fuzz-testing in an independent test-case, but not all at once.

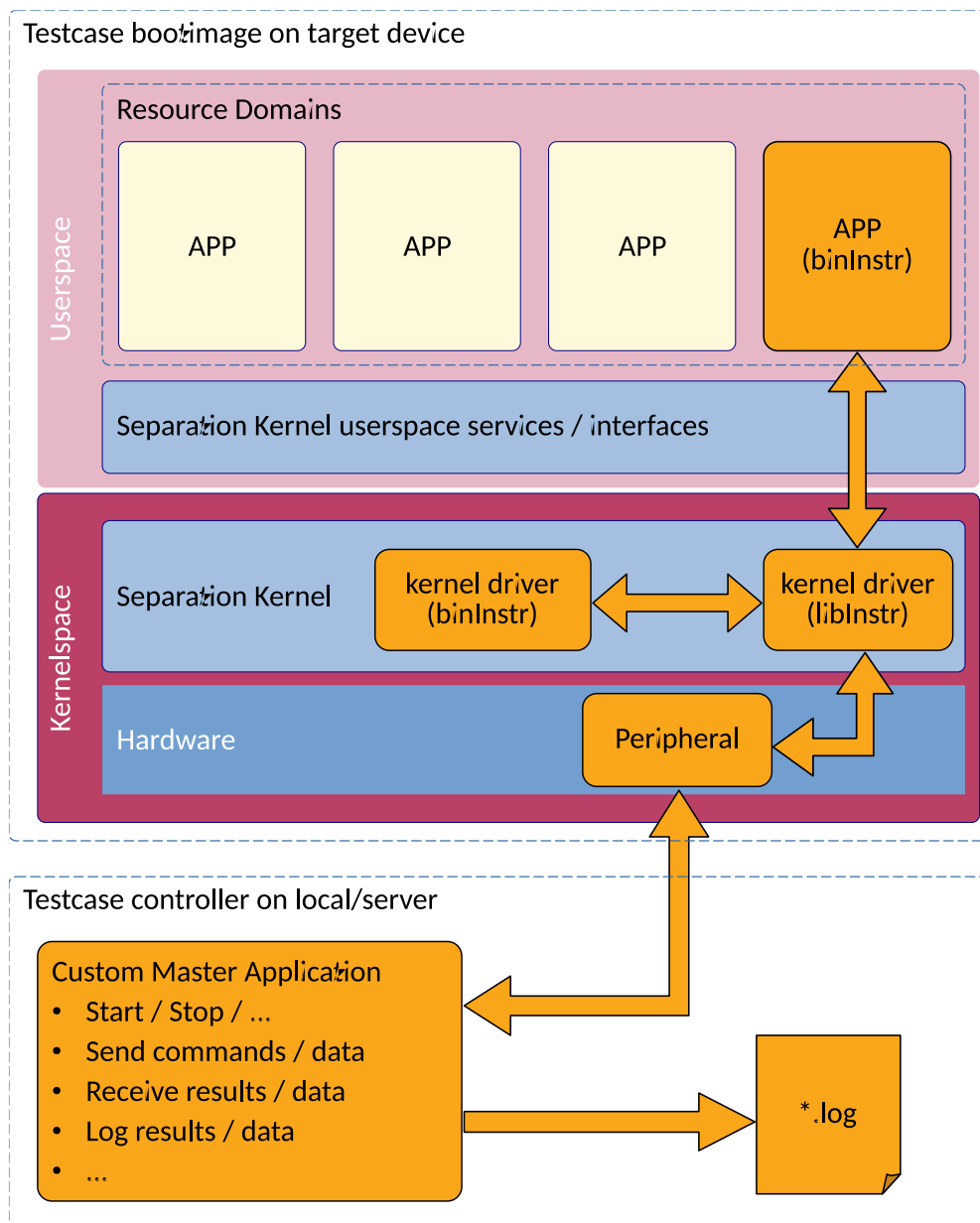


Figure 15 TF fuzzing integration project scheme



The test suite itself is activated via a parametrized make target, such as:

```
make clean all run TS=sec_tsuite TARGET=x86-64
```

This integration scheme can be assumed as general template for each tool flow that relies on instrumentation and dynamic execution on a hardware or emulated device. Therefore, any additional kind of testing can be added as new combination of test cases and test set to the test suite.

### 3.1.4 External Tool Integration for the TF

The TF supports the inclusion of external tools for additional analysis and evaluation scenarios as well via the Makefile integration approach. The data and control flow is shown in Figure 16. The needed tool setup and processing is encapsulated via dedicated make targets in a Makefile snippet that must be included in the parent Makefile.

At this point the external tools can be used by activating the provided make targets.

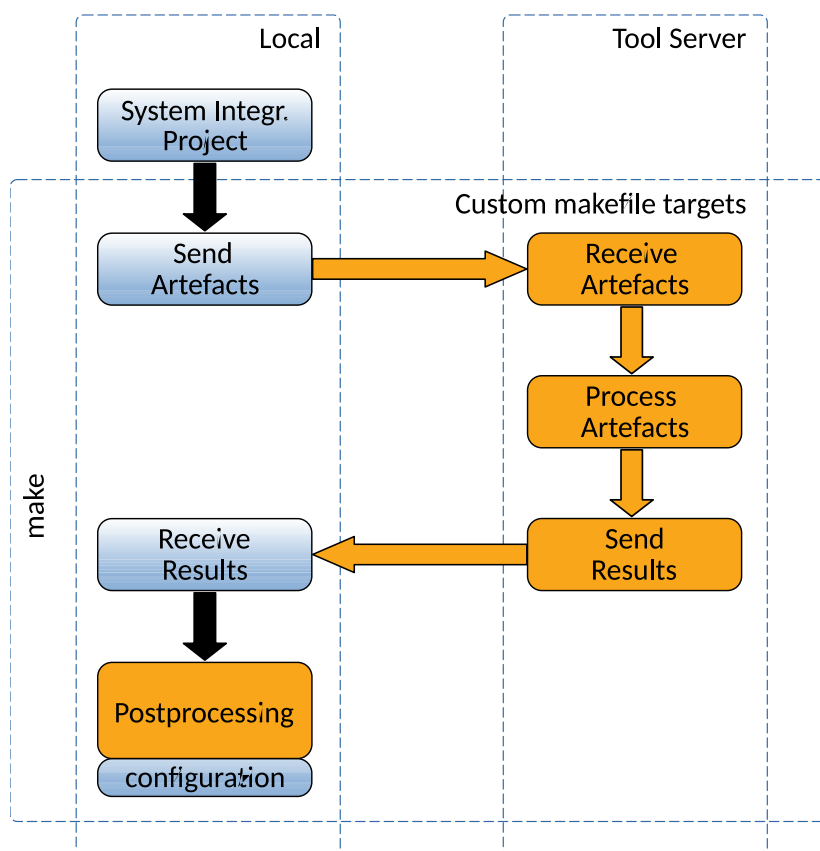


Figure 16 Information flow in automated server-based testing

Another use-case for security testing within the framework is the integration of the rule-checker of AbsInt. It executes source code compliance checks in the context of selected coding standard rules for the MISRA and the SEI CERT Secure C standards to achieve static code analysis.

The make targets that introduce the capabilities to interact with the rule-checker are introduced via:

```
include $(TOOLS_DIR)/rulechecker.mk
```

At this point the rule-checking procedure is performed via:

```
make clean all run-rulechecker
```

This command is sufficient to:

1. Send the source code files as well as the configuration files for the rules to be checked to the rule-checker on a server.
2. Let the rule-checker process the provided content.
3. Receive the analysis results from the rule-checker.

Depending on the rule strictness the test case will fail on unsecure code segments requiring to fix the source. A failed test case propagates to its test suite to halt the whole project-build process within the framework before it can be finalized, i.e., to form a release or a security patch.

## 3.2 Application TAS-Build-Chain

This chapter summarizes the proof-of-feasibility integration of kernel-component fuzz-testing within the TAS-Platform build-toolchain. Some of the commonalities with the separation kernel approach are described again.

### 3.2.1 *Structure and Components of the Testing Framework*

The fuzz-testing technology is based on the successful AFL framework, as it is for the former separation kernel approach. The core fuzzing engine is used unchanged. The communication channels are redirected from the testing host to the target using a remote communication proxy (RCP, also called “afl-wormhole”) on the testing side, and a fuzzing agent (FA) on the tested side. The RCP behaves as an AFL-instrumented user-space application on the testing host, i.e., the test framework supervisor machine. The Test-Agent (TA) injects the test payload into the target driver on the target system via a test-case specific interface. It also collects the coverage bits from the target driver and returns the coverage map back to the RCP. In normal operation the two components communicate through IP-network packets. In case of a crash of the test target, the networking sub-feature of the test-target is assumed to be rendered unfunctional. For successful coverage-based fuzzing, AFL also requires coverage information in case of a crash. This is achieved through a last-resort kernel-dump hook function to the serial console. Due to their low requirements, serial console outputs still work in many crash situations. Alternative debugging features, such as JTAG are oftentimes not available at this stage of system testing.

We have noticed, when a bug leading to endless recursion is fired, effectively depleting all stack memory, the console dump approach also fails, because it cannot be called. Recursion issues must be caught by static code analysis.

By returning the coverage map through the RCP to AFL, the AFL engine can use all its advanced algorithms to generate the next test payload. Crashes are logged by AFL on the testing host the usual way in the output folder. Obviously, since the tested target is a physical system, test-multi-threading cannot be used. AFL’s feature to run parallel coordinated instances on multiple hosts and targets was not trialled.

The RCP requires a GNU/Linux host as a runtime environment. The RCP currently has interfaces to access the TA implementation for the separation kernel (“pfuzz”) and for Linux (“fluzz”) from the same binary. For the separation kernel target, the RCP supports connecting to the tested target system through the development TCP data-channel multiplexer. For the Linux-kernel target, the RCP uses a plain TCP transport stream. The console needs to be redirected by external means onto a TCP socket for reading in a plain (e.g., telnet) style. This is, however, a common approach in larger testing facilities and readily available in the TAS-Testing Environment. If the target is run in a QEMU VM, the RCP can also access the QEMU control interface and make use of specific memory dumping functions to optimize test case execution.

The third component is required for preparation of the test target. To generate coverage information the target driver needs to be instrumented. Coverage information is hashed into a bitmap of configurable size between 1k and 64k byte. The original AFL approach uses injection of assembler statements at the beginning of each basic block.

A basic block is a sequence of linear statements. If a conditional branch or jump is passed, the compiler starts (two) new basic blocks, until the paths reunite or the function exits (see Figure 9 in chapter 2.2.4).

In contrast to the original approach by AFL, we have adapted an approach that uses a GCC plugin to inject GIMPLE statements (the intermediary representation language in GCC), see chapter 2.2.4. The plugin registers a handler that is called in one of intermediary compilation steps. The advantage is improved independence of the target architecture and inclusion in optimization steps. However, actual experimental comparison of using the plugin approach compared to the original AFL instrumentation technique on a user-space application have shown slightly slower performance for the plugin variant by a single digit percentage margin. (This was not a comprehensive benchmark.) The plugin is currently compatible with GCC version 5..9.

### 3.2.2 *Experimental Application*

The first step is to assure the GCC compiler for the target system has proper functioning plugin support. Experiences in the trial of the test-framework extension have shown that the shipped compilers for both platforms (TAS and separation kernel) did lack the support for various reasons (minimal required feature set, cross-compiler issues, dynamic library loading support, etc.) As an exact match of compiler version compiling the plugin and compiling the target is required, both compilers had to be recompiled to be usable for the fuzzing-security-testing framework extension.

While both recompilations of compilers succeeded in the end, it was always an unexpectedly large impediment, making the security test-framework extension anything but “plug-and-play”. Compilers are part of the target system toolchain qualification and assurance artefacts. The build environments for the compilers use special build systems themselves, which complicates their “minor” reconfiguration. This is especially true for cross compilers and specialized libc derivatives, applicable at the TAS-build-framework. As a result, compiler plugin support should be considered for inclusion in earlier system specification. For the proof-of-concept evaluation, out-of-automated-flow compilers were applied in a manual process.

In the second step, the target driver and its to-be-fuzzed interface must be selected. The current fuzzing framework extension has no sophisticated templating engine like Syzcaller to generically access type-based high-level defined interfaces. It is more suited for buffer inputs, such as protocol parsers, e.g., for Ethernet or CAN drivers. For the TAS platform, a special application level CAN protocol driver was selected. The selected interface must be manually coded into the TA module. The fluzz example provides a kernel-module exported symbol for a function call. The CAN protocol drivers provided a character device API.

To build the testable target image, first, the adapted TA must be included into the target image build process. Secondly, the GCC compilation flags must require the use of the plugin:

```
CFLAGS_target.o += -fplugin=path/to/afl-gcc-pass.so
```

To emphasize, *only* the compilation modules that are to be tested must contain the extra flag setting, not the TA, neither the kernel as a whole. Find the parallels to this step in the previous chapter for the separation kernel test-framework extension.

When the selected target driver is actually compiled, the plugin will output statistics related to its instrumentation activities. The sum of instrumented basic blocks may be a rough indication towards the coverage paths later found by AFL through guided fuzzing (see AFL’s GUI in ‘overall results’ box).

As the third step, AFL and afl-wormhole (the RCP) must be compiled. There are no requirements towards the compiler and the testing host may also be of different architecture than the tested target. Before starting AFL, the target should be up and waiting. The network connections, either the development TCP-multiplexer or plain IP should all be setup and the serial console should be reachable from the testing host.

When afl-wormhole is run on its own with command-line parameter '-h' it shows a short help screen, how to configure the target's / multiplexer's addresses and the target's timeout span. The latter is required to hang out for the reboot time after a crash of the target. When afl-wormhole is run as a pseudo-target in AFL, its outputs (stdout / stderr) are hidden. By default it logs its messages to a file in the same path named afl-wormhole.stderr.

Warning: By nature capturing the kernel output, the log file may become quite large and may not even provide insightful information to the tester. Crashes, chatty kernel consoles and broken test sessions may fill it up in bulk. It must be manually deleted.

If all parameters are correctly known, afl-wormhole can be started as the fuzzed binary of AFL, thus starting a test run. AFL, in most cases, fails to start with an error message, if things are not working as expected. In those cases afl-wormhole.stderr will typically provide further hints. Otherwise, follow AFL's UI:

```
[*] Checking core_pattern...
[*] Setting up output directories...
[*] Scanning 'i'...
[+] No auto-generated dictionary tokens to reuse.
[*] Creating hard links for all input files...
[*] Validating target binary...
[+] Persistent mode binary detected.
[+] Deferred forkserver binary detected.
[*] Attempting dry run with 'id:000000,orig:seed_file'...
[*] Spinning up the fork server...
[+] All right - fork server is up.
    len = 9, map size = 120, exec speed = 28615 us
[!] WARNING: Instrumentation output varies across runs.
[+] All test cases processed.

[!] WARNING: The target binary is pretty slow! See docs/perf_tips.txt.
[+] Here are some useful stats:

    Test case count : 1 favored, 1 variable, 1 total
    Bitmap range : 120 to 120 bits (average: 120.00 bits)
    data Exec timing : 28.6k to 28.6k us (average: 28.6k us)

[+] All set and ready to roll!
```

#### american fuzzy lop 2.52b (afl-wormhole)

process timing		overall results	
run time : 0 days, 0 hrs, 20 min, 32 sec		cycles done : 0	
last new path : 0 days, 0 hrs, 1 min, 51 sec		total paths : 111	
last uniq crash : none seen yet		uniq crashes : 0	
last uniq hang : none seen yet		uniq hangs : 0	
cycle progress		map coverage	
now processing : 77 (69.37%)		map density : 11.72% / 49.22%	
paths timed out : 0 (0.00%)		count coverage : 1.39 bits/tuple	
stage progress		findings in depth	
now trying : interest 16/8		favored paths : 35 (31.53%)	
stage execs : 157/247 (63.56%)		new edges on : 78 (70.27%)	
total execs : 43.8k		total crashes : 0 (0 unique)	
exec speed : 35.22/sec (slow!)		total tmouts : 0 (0 unique)	
fuzzing strategy yields		path geometry	
bit flips : 11/1568, 10/1544, 5/1496		levels : 3	
byte flips : 0/196, 0/172, 1/131		pending : 88	
arithmetics : 22/11.0k, 0/1085, 0/82		pend fav : 13	
known ints : 3/1031, 9/4437, 5/5477		own finds : 110	
dictionary : 0/0, 0/0, 0/386		imported : n/a	
havoc : 44/11.6k, 0/0		stability : 4.56%	
trim : 0.00%/40, 0.00%			

[cpu000: 30%]

Figure 17 Output of the robustness test run for the FiBuss protocol driver

**american fuzzy lop 2.52b (afl-wormhole)**

<b>process timing</b> run time : 0 days, 16 hrs, 35 min, 42 sec last new path : 0 days, 12 hrs, 40 min, 22 sec last uniq crash : 0 days, 12 hrs, 28 min, 1 sec last uniq hang : 0 days, 12 hrs, 27 min, 41 sec		<b>overall results</b> cycles done : <b>3</b> total paths : <b>253</b> uniq crashes : <b>1</b> uniq hangs : <b>1</b>
<b>cycle progress</b> now processing : 87* (34.39%) paths timed out : 5 (1.98%)	<b>map coverage</b> map density : 9.18% / 63.77% count coverage : 1.38 bits/tuple	
<b>stage progress</b> now trying : bitflip 2/1 stage execs : 14/71 (19.72%) total execs : 1.07M exec speed : 0.00/sec (zzzz...)	<b>findings in depth</b> favored paths : 42 (16.60%) new edges on : 190 (75.10%) total crashes : <b>1 (1 unique)</b> total tmouts : 2230 (1 unique)	
<b>fuzzing strategy yields</b> bit flips : 17/38.1k, 22/37.9k, 10/37.8k byte flips : 2/4753, 1/4676, 2/4540 arithmetics : 58/263k, 3/28.3k, 2/7493 known ints : 5/29.4k, 22/125k, 25/197k dictionary : 0/0, 0/0, 16/208k havoc : 68/70.6k, 0/0 trim : 0.00%/1981, 0.00%		<b>path geometry</b> levels : 15 pending : 176 pend fav : 0 own finds : 252 imported : n/a stability : <b>0.61%</b>

[cpu000: **29%**]

Figure 18 Final stats of the robustness test run for the FiBuss protocol driver

In the same manner as described for the separation kernel TF in the previous chapter, the kernel-module fuzzing extensions added to build-hooks of the TAS platform build-framework. However, the Yocto-Linux has different build process tools ( bitbake ), which is based on Python as the top layer of the framework but supports descending into Makefile build approaches as used in the certMILS security testing extension tools.

### 3.3 Results

The application of the low-level driver security testing framework was successfully examined on both MILS platforms of the certMILS project. As noted in the previous sections, there are multiple technical challenges related to the approach itself and the special requirements of the targeted systems.

Fuzz-testing requires many test runs to achieve results, so test runtime performance is of utmost importance. The additional components and network communication reduce the execution rate:

- The buggy test driver (with the tested core function being a cascaded string comparison) achieved an execution rate of ~5000 /s in a QEMU VM on the testing host.
- The same test case ran on a separate physical device achieved ~900 /s.
- Testing the CAN protocol driver for the TAS platform in their testing facility environment achieved ~40 /s.

While technically acceptable, the latter result is considered a slow test case. This may have different reasons and cannot be generalized in either direction.

## Chapter 4 Summary and Conclusion

The security requirements for MILS systems allow different techniques and methodologies for testing. In the introduction and, building on the results of previous deliverables, we explained the different attack surfaces for a MILS system. For example, we must assert that an application component (in a partition, see Figure 1) may be an attacker from a security point of view. There are also other attack vectors coming from the environment via the hardware interface, e.g., the network interface.

The robustness of MILS systems protecting the (minimal) assets computation time and memory have been proven with respect to the interfaces between the application components and the kernel. Tests related to these aspects were discussed in D4.1, and we have described integration into the separation kernel's TF and the TAS platform test framework, i.e., the functional testing requirements. However, kernel extensions required for specific privileged hardware interfaces or drivers with special timing requirements, are linked into the kernel-space and thus are not protected in the same way. As a result, kernel extensions must be integrated with appropriate care.

To compensate for technical limitations, we provide additional testing methodologies for kernel driver components and Linux kernel modules, the driver test-framework extension for security tests. By examining the approach, it was also shown (in case of the separation kernel) that it is also effective for multiple specialized security test tools, such as static code analysis, (static/dyn) API checking, etc. Due to attack surface-specific security test requirements and different tool applicability, we have not chosen a static one-fits-all package, but an extension-based plugin-approach to run security testing in proven test frameworks.

The separation kernel driver API checker has two parts to test conformance of interfaces. The test verifies the correct usage of the API between the driver and the kernel and vice versa. It can also scan for allowed and forbidden API calls and trace invalid access to kernel-internal resources on address bases, i.e., also for black-box driver objects. Traces of the API calls are collected at (test-) run-time and identified for calling in correct order. The second part considers the interfaces from the kernel to the driver, e.g., the kernel driver callback API, and fills the data payloads with random data to test for robustness of the driver. This approach uses state-of-the-art fuzzing techniques to optimize code-coverage and testing performance. This second part was also trialled on the TAS-platform to assert its flexibility within the MILS approach.

We demonstrated the testing methodologies using sample drivers and a protocol driver to show the test-framework's effectiveness in identifying security and robustness issues in kernel device drivers.

We focused on improving the solution for best balance between ease of application by different (developer) target groups, architecture independence and testing performance. The result is a test framework extension for security tests that can add confidence to the correctness of kernel extensions. The framework extension is also useful to uphold the confidence in case of patching or updating of an installed system.

Overall, the security-testing framework extension approach is compliant to IEC 62443 and CC requirements. Including other test tools, e.g., the external separation kernel API-Fuzzer (see D4.1, robust APIs), commercial tools like Nessus for EDSA compliance, as well as the static analysis tools from D1.2 (e.g., AbsInt rule-checker), the framework can provide coverage for a large area of security testing requirements.



## Chapter 5 List of Abbreviations

Abbreviation	Translation
API	Application Programming Interface
ATE	CC security assurance class “Tests”
CC	Common Criteria for Information Technology Security Evaluation (CC)
EAL	Evaluation Assurance Level
EDSA	Embedded Devices Security Assurance
GCC	GNU Compiler Collection
HW	Hardware
IACS	Industrial Automation and Control System
IEC	International Electrotechnical Commission
kAFL	Kernel AFL
KVM	Kernel Virtualization Module
OS	Operating System
PT	Intel Processor Trace
QEMU	The Q System Emulator
RCP	Remote Communication Proxy
SK	Separation kernel
ST	Security Target
TF	Test Framework
TSF	Target of Evaluation Security Functionality
TSFI	Target of Evaluation Security Functionality Interface

## Chapter 6 Bibliography

- [1] T. Schulz, A. Hohenegger, S. Persson, A. Ortega, R. Hametner, M. Paulitsch, C. Gries, S. Tverdyshev, H. Blasum und K. Tomáš, „Security testing framework: strategy and approach (certMILS D4.1),“ September 2017. [Online]. Available: <https://zenodo.org/record/2586591>.
- [2] A. Hohenegger, „Die Common Criteria und IEC-62443,“ in *Deutscher IT-Sicherheitskongress*, 2019.
- [3] CENELEC, prTS 50701: Railway applications – Cybersecurity, draft version D7E3, 2019.
- [4] 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>.
- [5] ISA Security Compliance Institute, “ISASecure - IEC 62443-4-1 - SDLA Certification,” 2017. [Online]. Available: <http://www.isasecure.org/en-US/Certification/IEC-62443-SDLA-Certification>.
- [6] J. Rollo, A. Alvarez de Sotomayor, B. Caracuel, A. Ortega, R. Hametner, S. Tverdyshev, H. Blasum, T. Kertis, O. Havle, T. Schulz und M. Hager, „List of tools and techniques applicable for high and medium assurance for efficient assurance (certMILS D1.2),“ December 2017. [Online]. Available: <https://zenodo.org/record/2586480>.
- [7] J. Reinders, “Intel Software Developer Zone: Processor Tracing,” Intel Corporation, 2013. [Online]. Available: <https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing>. [Accessed September 2017].
- [8] ARM, “ARM CoreSight Datasheet,” 2015. [Online]. Available: [https://www.arm.com/files/pdf/CoreSight\\_Datasheet.pdf](https://www.arm.com/files/pdf/CoreSight_Datasheet.pdf). [Accessed 2017].
- [9] M. Zalewski, “american fuzzy lop,” [Online]. Available: <http://lcamtuf.coredump.cx/afl/>. [Accessed 2017].
- [10] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel and T. Holz, “kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels,” in *USENIX Security Symposium*, 2017.
- [11] L. Chien, “Intro to Compiler Development,” Mediatek, 2018. [Online]. Available: <http://slide.logan.tw/compiler-intro/>. [Accessed 2018].
- [12] “ISO/IEC 15408-1:2009, Information technology - Security techniques - Evaluation criteria for IT security - Part 1: Introduction and general model,” 2017. [Online]. Available: <https://www.iso.org/standard/50341.html>.
- [13] 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>.