SAFURE

D3.3
Integrity Methodology

Project number:	644080
Project acronym:	**SAFURE**
Project title:	SAFety and secURity by dEsign for interconnected mixed-critical cyber-physical systems
Project Start Date:	1st February, 2015
Duration:	36 months
Programme:	H2020-ICT-2014-1
Deliverable Type:	Report
Reference Number:	ICT-644080-D3.3 / 1.0
Work Package:	WP 3
Due Date:	July 2017 - M30
Actual Submission Date:	31st July, 2017
Responsible Organisation:	SYM
Editor:	Jonas Diemer
Dissemination Level:	PU
Revision:	1.0
Abstract:	The report will provide the design guidelines for ensuring the integrity of safe and secure systems based on the analysis methods and protection mechanisms developed in WP3.
Keywords:	Algorithms, Mixed-Criticality, Temperature, Data integrity, Timing integrity, Resource sharing integrity

This project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 644080.

This work is supported (also) by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 15.0025. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the Swiss Government.

Editor

Jonas Diemer (SYM)

Contributors / Reviewers (ordered according to beneficiary numbers)

Martin Deutschmann (TEC)
André Osterhues (ESCR)
Mikalai Krasikau (SYSG)
Jonas Diemer, Björn Gebhardt (SYM)
Sylvain Girbal (TRT)
Gabriel Fernandez, Jaume Abella, Francisco J. Cazorla (BSC)
Rehan Ahmed, Philipp Miedl, Lothar Thiele (ETHZ)

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 users thereof use the information at their sole risk and liability.

Executive Summary

Today's networks and embedded systems are becoming increasingly complex. Thus, also the error-proneness of such systems is growing. To prevent failures and to increase the safety, methodologies for design, regarding thermal influences, data integrity and timing, are required. To reach this goal, the SAFURE partners have developed desgin guidelines for developing strategies and introduce some tools to guide architects and developers of such systems.
This document describes the integrity methodology developed in SAFURE. Specifically, it presents integrity methodologies for multicores and networks, for which it shows how various tools and mechanisms developed in this project can be used together to ensure the safety and security of embedded systems. These methodologies aim at temperature and resource sharing integrity, for data integrity, the SAFURE project refers to existing methodologies in the design guidelines given in deliverable D3.2.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

The complexity of networks and embedded systems is continuously increasing, and with the same speed also the error-proneness of such systems is growing. To prevent failures and to increase the safety, methodologies for design, regarding thermal influences, data integrity and timing, are required. To reach this goal, the SAFURE partners have developed design guidelines for developing strategies and introduce some tools to guide architects and developers of such systems.

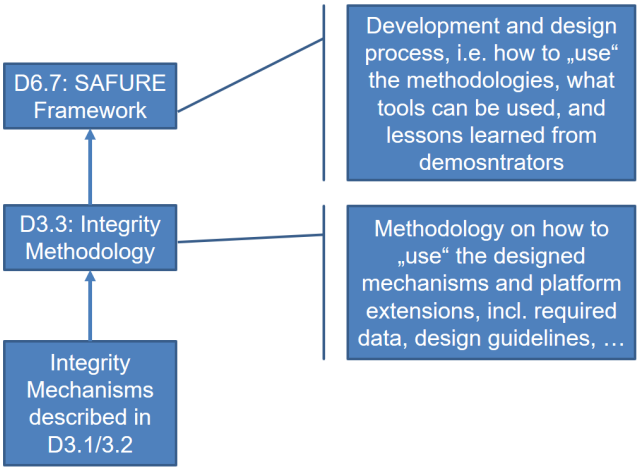

Figure 1.1: Overview of the relationship of D3.3 with other deliverables

This document presents an integrity methodology consisting of several of these guidelines, which utilize mechanisms developed in SAFURE. As illustrated in Figure 1.1, it builds upon the integrity mechanisms developed and presented in deliverables D3.1 and D3.2 and focuses on how to apply them in the development of safe and secure systems. This methodlogy is a central component of the SAFURE framework described in D6.7 which ties together the methodologies and also presents lessons learned from their application in the demonstrators of WP6.
Chapter 2 presents an integrity methodology for multicores. Specifically, guidelines and tools for temperature integrity are presented in Section 2.1. It concentrates on how to construct a thermal model of multi core systems and explains how to apply a thermal isolation server scheme to provide thermal protection to a mixed-critical application executed on such multi-core systems. At last this chapter gives a quantification of the security threats posed by temperature and frequency covert channels and offers possible mitigation strategies. Furthermore, Section 2.2 presents guidelines to detect vulnerabilities originating from shared resources (such as busses and caches) on the execution times of the applications. Then, a methodology for measuring the shared resource requirements of applications, which can then be used as an input to budgeting of resources. Section 2.3 presents a performance measurement environment for characterizing application performance, target platform behavior and timing interferences.
Chapter 3 focuses on an integrity methodology for networks. For Ethernet networks, a timing analysis methodology is presented in Section 3.1 that uses a network model to analyze contention effects in the network to verify that the network (and its configuration) still meets the timing requirements even in the case of errors (such as babbling idiots, i.e. faulty nodes that disturb the network). Finally, a

methodology to minimize possible timing covert channels over a network is presented in Section 3.2 using virtual private networking (VPN) and PikeOS.
Within the SAFURE consortium, is was decided not to design new data integrity algorithms, because this should be left to cryptographic experts. Therefore, there is also no new data integrity *methodology* to be presented in this deliverable. For design guidelines, please refer to deliverable D3.2, which includes a discussion on the suitability of the existing algorithms for different embedded scenarios.

Chapter 2 Integrity Methodology for Multicores

2.1 Design guidelines and tools to ensure Temperature Integrity

This chapter details how the thermal protection mechanism explained in SAFURE deliverable D3.2 can be applied in practice. Furthermore, this section also details the quantification of security risk posed by temperature and temperature control mechanisms. Possible mitigation actions for countering this security threat are also proposed.

2.1.1 Methodology and Design Guidelines for Thermal Modeling

Several research works derive thermal models of the architectural platform [26, 25, 19, 7]. The approach followed in SAFURE is similar to the calibration based approach used in [25].

2.1.1.1 Thermal calibration tests

With the thermal calibration tests, we intend to capture the following effects:

- Transient temperature characteristics of each core.
- Steady state temperature characteristics of each core.
- Steady state temperature effect of a given core on every other core.

Note that we do not need to characterize transient thermal impact of a given core on every other core since this characterization is not required by the Thermal Isolation Server (TIS) scheme. The thermal calibration tests were performed on a quad-core Core i7-4700MQ platform. Following are the specifications of the performed calibration tests:

1. Periodic task with period of $10s$ and computation time of $5s$ running on core $i \in \{1, 2, 3\}$. All other cores idle. Test duration $60s$.
2. Core i active for 40 minutes where $i \in \{1, 2, 3\}$. All other cores idle. This is followed by all cores idle for 20 minutes.
3. Cores (i, j) active for 40 minutes where $(i, j) \in \{(1, 2), (1, 3), (2, 3)\}$. All other cores idle. This is followed by all cores idle for 20 minutes.

In total, this entails 9 calibration tests. In our evaluations, core 0 was not thermally modeled. To get the thermal model, the calibration traces were normalized to remove any DC offset. Normalized traces of type 1 and 2 for a given core were concatenated to provide a larger trace that captured both transient and steady state characteristics of a given core. The temperature transfer function of each core was then estimated by employing Matlab's tfest function. The steady state temperature effect of a given core on other cores was inferred directly from calibration traces of type 3.

Purpose	CL	Count	Period (ms)	Exec.Time (ms)
Sensor data acquisition	HI	5	200	10
Localization	HI	3	200	10
	HI	3	1000	50
	HI	1	5000	50
Flightplan management	HI	4	1000	50
	LO	4	1000	50
Trajectory computation	HI	2	1000	50
	HI	1	5000	750
	HI	1	5000	180
	HI	1	5000	150
	HI	1	5000	90
	HI	1	5000	75
Guidance	HI	1	200	10
Nearest Airport	LO	1	1000	50

Table 2.1: FMS Parameters

2.1.2 Methodology and Design Guidelines for Thermal Protection Mechanisms

In this section we illustrate how thermal protection can be achieved in a mixed-critical setting by using Thermal Isolation servers explained in SAFURE deliverable D3.2 section 2.1. To illustrate this, we consider a dual-critical flight management system application (FMS) [12]. The same FMS application is also used in the Work Package 4 prototype. It comprises critical tasks such as localization and guidance. It also comprises of lower criticality tasks such as flightplan management and a task that determines the nearest airport. The parameters of all tasks are given in Table 2.1. All tasks are assumed to be periodic with implicit deadlines. The goal here is to provide the HI criticality application thermal protection.

2.1.2.1 Platform and experimental setup

We emulate the FMS application on a hardware test-bed. For this purpose, we execute tasks with parameters given in Table 2.1 on a laptop platform (Lenovo Thinkpad T440p). The platform has a Core i7-4700MQ quadcore processor. The operating frequency of all cores is fixed to 3.2 GHz and the fan speed is set to maximum. Our hardware platform is running Ubuntu 16.04 with preempt-rt patch [11]. This patch makes the Linux kernel preemptable, leading to near real-time performance. To execute the tasks, we augmented the HSF [28] framework by adding a new scheduling class for TIS and adding the ability to log on-chip thermal sensors. Core 0 is reserved for HSF scheduling and temperature logging tasks. In all of the following results, the temperature of core 0 is not plotted since it does execute any tasks of the FMS application. LO criticality tasks are executed on core 1 using Earliest Deadline First (EDF). HI criticality tasks are executed on core 2 and core 3 using TIS. The Thermal constraint is violated if the temperature of any core exceeds 70° C. We use different scheduling algorithms for LO and HI criticality applications to demonstrate how Thermal Isolation Servers can be combined with other scheduling algorithms, to provide thermal protection.

2.1.2.2 Computing available thermal budget:

To provide thermal protection to the HI criticality tasks, we need to compute the available thermal budget for execution of HI criticality tasks using TISs, and verify that this budget is sufficient. To compute this budget, we perform the following calibration tests:

1. Test1: Core 1 executing LO criticality application using EDF. Core 2 and core 3 always idle.

2. Test2: Core 1 executing LO criticality application using EDF. Core 2 and core 3 always active.

3. Test3: Core 1 always idle. Core 2 and core 3 always active.

In each of these tests, we record the temperature of all cores . Each test is sufficiently long such that temperature profile is similar across hyperperiods[1]. $\boldsymbol{T}(t, \text{Testx})$ is used to represent the temperature profile of test $x \in \{1, 2, 3\}$. Furthermore, $\boldsymbol{P}X(\boldsymbol{T}(t, \text{Testx}))$ is a vector representing the X^{th} percentile temperature of each core, e.g. $\boldsymbol{P}100(\boldsymbol{T}(t, \text{Test1}))$ represents the maximum temperature of each core in Test1. $\boldsymbol{T}^{\infty}(\boldsymbol{B}^{\mathbf{idle}}) = [36.8, 38.12, 38.6]^{\intercal}$ and is calculated by averaging several temperature values when each core is idle. The total available thermal budget of cores 1-3 (represented as $\boldsymbol{\Lambda}^{\mathbf{total}}$) is computed as:

$$\boldsymbol{T}^{\boldsymbol{\Delta}} - \begin{bmatrix} P99.9(\boldsymbol{T}(t, \text{Test1}))_1 \\ P99.9(\boldsymbol{T}(t, \text{Test2}))_2 - P99.9(\boldsymbol{T}(t, \text{Test3}))_2 + [\boldsymbol{T}^{\infty}(\boldsymbol{B}^{\mathbf{idle}})]_2 \\ P99.9(\boldsymbol{T}(t, \text{Test2}))_3 - P99.9(\boldsymbol{T}(t, \text{Test3}))_3 + [\boldsymbol{T}^{\infty}(\boldsymbol{B}^{\mathbf{idle}})]_3 \end{bmatrix} = \begin{bmatrix} 16.0 \\ 28.88 \\ 27.4 \end{bmatrix}$$

99.9 percentile values are used to account for outliers in temperature measurements.

2.1.2.3 Applying thermal calibration tests

To apply TIS, we need to first develop a thermal model of the platform. To partition tasks in a thermally optimal manner (according to SAFURE deliverable D3.2 section 2.1.6.1) we only require $-\boldsymbol{A}^{-1} \cdot \boldsymbol{C}^{-1} \cdot \psi^d$ value to estimate the temperature[2]. $-A^{-1}_{i,j} \cdot \psi^d / C_{i,i}$ can be interpreted as the steady state temperature of core i when core j is dissipating ψ^d and ambient temperature is 0. These steady state values can be acquired directly from calibration tests of type 2 and 3. $\boldsymbol{T^c}(y)$ is used to represent the temperature vector at the end of the calibration test trace where all core/s in y are active. $-\boldsymbol{A}^{-1} \cdot \boldsymbol{C}^{-1} \cdot \psi^d$ can be evaluated as:

$$\begin{bmatrix} T^c(1)_1 - [\boldsymbol{T}^{\infty}(\boldsymbol{B}^{\mathbf{idle}})]_1 & T^c(1,2)_1 - T^c(1)_1 & T^c(1,3)_1 - T^c(1)_1 \\ T^c(1,2)_2 - T^c(2)_2 & T^c(2)_2 - [\boldsymbol{T}^{\infty}(\boldsymbol{B}^{\mathbf{idle}})]_2 & T^c(2,3)_2 - T^c(2)_2 \\ T^c(1,3)_3 - T^c(3)_3 & T^c(2,3)_3 - T^c(3)_3 & T^c(3)_3 - [\boldsymbol{T}^{\infty}(\boldsymbol{B}^{\mathbf{idle}})]_3 \end{bmatrix} = \begin{bmatrix} 27.2 & 9.48 & 6.80 \\ 8.68 & 21.60 & 10.68 \\ 7.00 & 8.4 & 25.8 \end{bmatrix}$$

For a detailed description of the thermal model please refer to SAFURE deliverable D3.2 section 2.1.1.1.

2.1.2.4 Server configuration and results:

We can now do thermal constrained task partitioning to assign the HI criticality tasks to core 2 and core 3; as the LO criticality tasks are scheduled using EDF on core 1. Furthermore, an additional constraint is added to the task partitioning formulation which prevents any HI criticality task from being assigned to core 1; since core 1 is reserved exclusively for LO criticality application.
The utilization of core 2 and core 3 after partitioning is 0.678 and 0.431 respectively. In our simulations, we computed a safe bound on ϵ (TIS preemption overhead) to be $150\mu s$. Based on these values, we chose the following server configurations to execute HI criticality tasks:
$S_1 : P_1 = 10ms, U_1 = 0.693, \text{Core} = 2$
$S_2 : P_1 = 10ms, U_1 = 0.546, \text{Core} = 3$
These server configurations lead to a substantial reduction in the peak system temperature. Fig. 2.1 shows the system temperature when the proposed scheme is used. Fig. 2.2 shows the system temperature when the HI criticality tasks are partitioned on cores 2 and 3 using worst-fit bin packing, and tasks are executed using EDF. The later case results in violation of the thermal constraint as core 3 reaches a peak temperature of $72°C$. For the TIS based scheme, the temperature always remains below $64°C$. Fig. 2.1 also shows the upper bounds on peak temperature.

[1] hyperperiod is the least common multiple of the periods of all tasks in a given task-set
[2] Description of all thermal model parameters is given in SAFURE deliverable D3.2 section 2.1.1.1

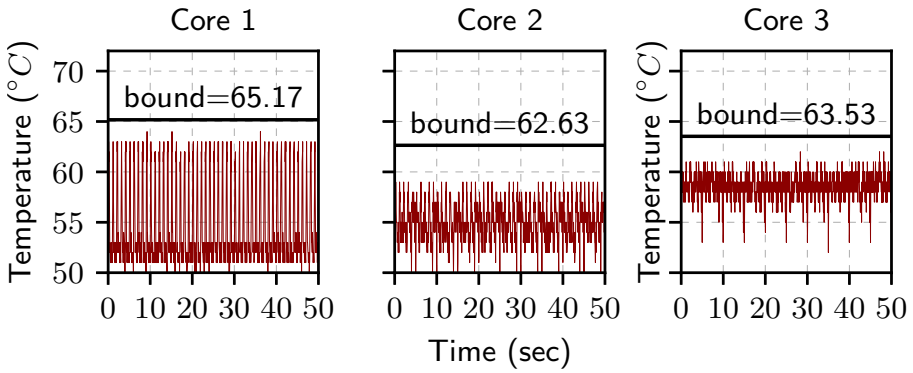

Figure 2.1: System temperature. HI criticality tasks partitioned using proposed scheme and scheduled using TIS

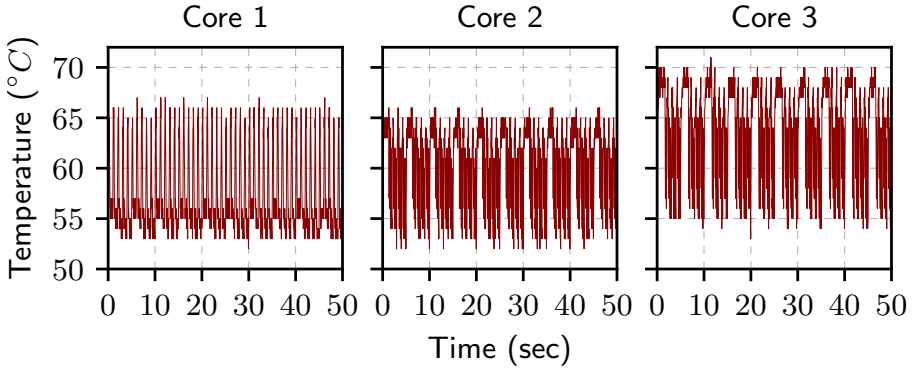

Figure 2.2: System temperature. HI criticality tasks partitioned using worst-fit bin packing and scheduled using EDF

To compute temperature bounds, we simulate the server schedule using the temperature transfer function we evaluated in section 2.1.1.1 until a steady state is reached (temperature is same across different periods of TIS). Λ_2^1 and Λ_3^2 are the self-core thermal budgets of S_1 and S_2, respectively. These can be computed directly from the thermal simulation of the corresponding server schedule. Non-self-core thermal budgets are computed as:

$$\Lambda_j^1 = \Lambda_2^1 \cdot A_{j,2}^{-1}/A_{2,2}^{-1} \quad j \in \{1,3\}$$
$$\Lambda_j^2 = \Lambda_3^2 \cdot A_{j,3}^{-1}/A_{3,3}^{-1} \quad j \in \{1,2\}$$

The upper-bound on peak temperature can be computed using $\boldsymbol{T^{\Delta}} - \boldsymbol{\Lambda^{\text{total}}} + \boldsymbol{\Lambda^1} + \boldsymbol{\Lambda^2} + \boldsymbol{\delta}$, where $\boldsymbol{\delta}$ is an error margin of 1°C which is added to account for the limited precision of the on-chip thermal sensors.

2.1.3 Methodology and Design Guidelines for Quantification of Thermal Related Security Risks

In its *Orange Book* [31], the US Department of Defense stated that trusted computing environments should have the capability to audit covert channels with bandwidths of more than 0.1 bps and that in most application environments, covert channels with a bandwidth of at maximum 1 bps are acceptable. Furthermore, we urge that in a threat analysis, also the small message criterion by Moskowitz and Kang [20] is considered. The small message criterion suggests that the capacity alone is not sufficient to quantify the threat potential of covert channels, but also the security level and size of the leaked data. Therefore we advise designers to analyse exposed covert channels not only using the *Orange Book* but also the *small message criterion*. This can be done within a scenario similar to the

following: The highly sensitive data is a cryptographic key which could be used to enter a company network or servers. The cryptographic key is saved on the hand-held device or a laptop of a company, such that the employee can access the company network and servers while being on a business trip. Today, the National Institute of Standards and Technology considers elliptic-curve cryptography with a key size of 512 bit to be highly secure [4]. Now, assume that an attacker manages to deploy a setup of one of the covert channels on the hand-held device.

2.1.3.1 Thermal Covert Channel

Based on our results, we cannot exclude the possibility that these channels might be a security issue, as the capacity could be in the order of 300 bps for the same-core channel. We presented a transmission scheme based on Manchester encoding that sensibly improves the performance of previous work and we studied the sensitivity of our results to non-ideal conditions. With this scheme, we were able to achieve rates of more than 45 bps on the same-core channel and more than 5 bps on the 1-hop channel, with less than 1% error probability. Furthermore, in a realistic attack scenario we show an average goodput of 1.358 bps. Considering the scenario defined above, an attacker can be capable to leak the cryptographic key in less than 7 minutes with a very simple implementation of the covert channel. Therefore the thermal covert channel poses a not negligible threat to a secure system.
As we reported in deliverable D3.1, the on-chip temperature sensors that enable the thermal covert channels we studied are easily accessible by user-level apps on current mobile systems. A technically simple way to block the potential threats coming from these channels is to restrict access to the temperature sensors to trusted code. If temperature information needs to be made available to user apps (e.g., a CPU temperature monitor), viable mitigation strategies include increasing the refresh interval from milliseconds to seconds or minutes and reducing the sensor resolution, thus directly limiting the capacity of the thermal covert channels. While mitigating this threat is not technically challenging, it requires shipping security patches to a huge base of affected devices running different versions of different system software stacks. Our aim with this work was building awareness on the potential threat that current systems are exposed to and providing a quantitative study that can be used as a base to decide what actions to take in order to mitigate this threat.

2.1.3.2 Frequency Covert Channel

The slowest implementation of the frequency covert channel, presented in deliverable D3.2, yields a throughput of 0.90 bps, which seems negligible. However, considering our threat model and the scenario defined above, the attack can be carried out while the device is not actively used, i. e. during the night when the employee is sleeping. Assuming a short night with 5 hours, the attacker is able to leak 16200 bits; which is enough bandwidth to be able to leak a 512 bit key and apply error detection and correction to ensure an error-free transmission. Furthermore, as our implementation and transmission scheme are very simple, this attack can be executed more efficiently. Using this reasoning and considering that

- the attacker does not need any special permissions to establish the frequency covert channel,
- almost every current mobile multicore system is affected,
- systems can often be compromised by leaking relatively small amount of information, i. e. a cryptographic key or a password, and
- these systems are often idle, which makes the execution of the attack easier,

we can state that the covert channel based on the frequency of the core needs special attention by device and OS manufacturers.

During their execution, applications can commonly require access to timers or to timestamps. With these timers and timestamps, applications are able to indirectly measure processing speed without the need for special OS permissions. As a result, there is no simple possibility to fully close the security gap we analyzed. Still, there are mitigation strategies that can be used during design and runtime of a device to minimize the threat that emanates from the frequency covert channel.
One possible hardware measure is not to share the frequency domain among different cores and to not execute applications with different security clearances on the same core. Devices can reserve one core of the system, which does not share its frequency domain with any other core, for executing all applications with access to restricted data, while all non-secure applications are executed on cores in some other frequency domain.
Another possible solution are malware detectors, which use different characteristics of a process execution, i. e. memory access or instruction stream, and apply classifiers to identify malicious processes. The malware detectors can either be realised in software using middleware or system calls [24, 5, 2], or as HMDs [13, 3, 23, 36, 6] running inside a hardware-based enclave or directly in hardware. Using training frameworks, for example for HMDs [15], some existing malware detectors could be trained to detect the frequency covert channel.
Another approach is to alter the part of the system which is responsible for the information leakage [27]. To do so, the mitigation strategies have to be included into the governor algorithms and implementations. Therefore, a smart governor design can reduce the available bandwidth of the frequency covert channel such that the threat is negligible. However, this strategy needs to be evaluated carefully as it can also have a negative impact on the effectiveness of the governor in terms of energy saving.
Finally, attacks only work well on idle devices, i.e. when there is no additional load that influences the core frequency. Detecting and protecting the idle-mode could be used correspondingly. For example, in case of smartphones or tablets, the governor design could take device characteristics into account, i.e., controlling the idle mode depending on whether the screen is on or off. An example is the *InteractiveX* governor for Android[3].

2.2 Guidelines and Strategy for Vulnerability Detection for Multi-Cores

As described in D3.1 and D3.2, vulnerability detection for multi-cores builds upon the concepts of signatures and templates. Signatures describe succinctly how much the task under analysis (TuA) accesses shared resources (number and type of accesses). Analogously, templates describe how much contenders are allowed to access those shared resources (number and type of accesses) so that their contention can be properly upper-bounded.
In order to use this approach, a number of steps need to be followed (see Figure 2.3), which we list next:

1. Identify relevant shared resources in the multi-core.
2. Identify relevant events creating contention in those shared resources.
3. Quantify the impact on contention of each event in each shared resource.
4. Get Worst-Case Execution Time (WCET) bounds based on signatures/templates that can be used for scheduling purposes and by the hypervisor.

The remainder of this section provides details on the application of those steps.

[3] http://androidforums.com/threads/android-cpu-governors-explained.513426/

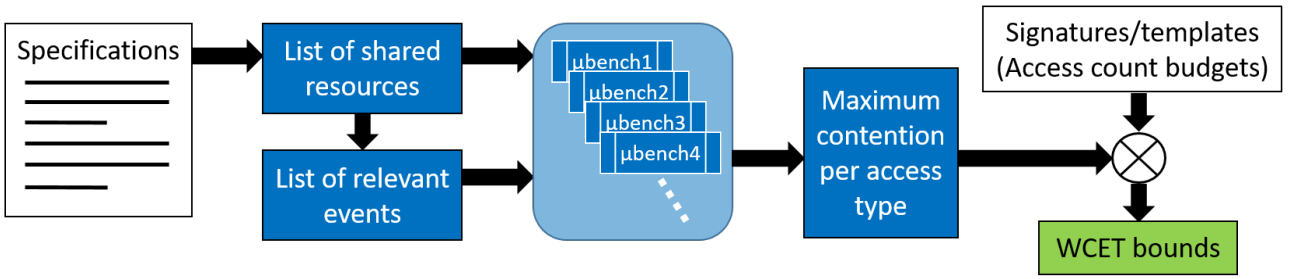

Figure 2.3: Steps for the assessment of vulnerability detection for multi-cores.

2.2.1 Identification of Shared Resources

Shared resources cannot be identified with a black-box approach in general. Instead, one needs to resort to the processor specification and review its architecture to identify potential sources of interference across cores in the multi-core under consideration. This tedious step is unavoidable unless the chip vendor provides explicitly a list of those resources, which is not often the case.
Unfortunately, such documentation is not guaranteed to be either accurate or complete. Moreover, the identification of shared resources is potentially an error-prone process, especially given that documentation for many embedded processors is in the order of some thousands of pages. For instance, some parts of the Freescale P4080 specification have 2,000 pages [9]. Similarly, the Infineon XMC4500 microcontroller documentation has more than 2,500 pages [14].
Therefore, while it may be easy to identify some resources with large impact in execution time, such as shared buses, shared cache memories and shared memory controllers, it is hard to claim the completeness of this step. In particular, some shared buffers or queues may not be conveniently documented in the specification and may only be discovered empirically when controlled experiments to quantify the impact of contention on shared resources (see Section 2.2.3) reveal unexpected timing behaviour.

2.2.2 Identification of Relevant Events

Once a list of shared resources has been elaborated, then it is required to identify the different types of events that may lead to some form of contention for each such shared resource. As for the identification of the shared resources, hardware experts may identify the main types of events causing contention, but claiming completeness is hard to sustain. Instead, empirical data obtained with controlled experiments is the only practical evidence in this direction as long as results match expectations.
The main types of accesses to be considered in the access to shared resources include, at least, the following ones:

- Read and write accesses.
- Cache line misses and evictions (so long read/write accesses).
- Dirty data evictions from caches.
- Atomic operations, which may retain shared resources busy during arbitrarily long periods.
- Snoops, invalidations and other types of activities related to the coherence protocol.
- Uncacheable accesses (often related to I/O activities).

2.2.3 Quantification of Contention Impact

Parameters such as the maximum latency of the different events in shared resources are seldom available in processor specifications. For instance, in the context of an avionics study in a Freescale P4080 processor, static timing analysis needed to build upon measurements to upper-bound the latency of accesses to shared resources [21].
Whenever information cannot be obtained from the specifications, we can only resort to empirical evidence to approximate those bounds. For that purpose, we need to devise micro-benchmarks placing maximum contention on shared resources and measure how the execution time of the TuA – which is modelled by another micro-benchmark to maximize contention experienced – increases due to contention. For instance, as it will be described in D4.2, appropriate micro-benchmarks can be built for both, the TuA and contenders, so that maximum contention is triggered systematically in round-robin and FIFO arbitrated shared resources. Details on these techniques have already been published by project partners within the scope of SAFURE [8]. These micro-benchmarks can be implemented with instructions that trigger the different types of events described before for each shared resource.

2.2.4 WCET Impact of Contention and Usage of those Bounds

Once we have determined the number and type of accesses to each shared resource experienced by the TuA, as well as the maximum contention that each such access type may experience, then we only need to choose the amount of contention allowed in shared resources. Different approaches can be followed for this: (1) fully time-composable (fTC) bounds or (2) partially time-composable (pTC) bounds.
The former (fTC bounds) account for the maximum contention that could ever be experienced by the TuA. This requires assuming that contenders will perform an arbitrarily large number of requests to each shared resource, being such requests of the type that generates the highest contention in each shared resource. This approach, referred to as fTC, allows deriving time bounds that hold upon integration with *any* other tasks in the other cores. However, this may be unrealistic and extremely pessimistic.
Instead, pTC bounds account for a given level of contention. Such level of contention is provided in the form of access counts of each type, and thus, as explained in D3.2, it is a matter of accounting for the maximum contention that those accesses can produce on the TuA. Deriving these access counts can be done in several ways: the software integrator can provide those counts to the different software providers in the form of *budgets*, so that each software provider adheres to its budget. Then, contention bounds will hold. Alternatively, each software provider can develop its software components unaware of the budget and let the integrator estimate the number of requests of each task, so that contention impact on each other task can be estimated.
Finally, if pTC bounds are used, there is a risk of one task exceeding its budget, thus producing more contention than budgeted, so that there is some risk of overruns for other tasks. This should not be allowed, especially in the context of mixed-criticality systems. Therefore, the hypervisor needs to monitor the number and type of accesses of the different tasks to the shared resources by means of the Performance Monitoring Counters (PMCs) in the platform. In case of a task exceeding its budget, then it is up to the hypervisor to take the appropriate corrective actions (e.g. stopping the violating task, moving the system to a safe state).

2.3 METrICS: a Measure Environment for Multi-Core Time Critical Systems

This section describes the performance measurement environment called METrICS (Measure Environment for Time-Critical Systems), developed at TRT. The purpose of this measurement environment is: 1) to allow an accurate estimation of embedded application performance. 2) to characterize the application behavior on the target hardware platform. 3) to integrate into a PikeOS real-time environment. 4) to quantify timing interference inherent to mutli-core platforms.
This implementation of METrICS has been developed for PikeOS 4.1, and the hardware target is the ARM Juno board.

2.3.1 Purpose & Requirements

The objective of the METrICS measure environment is to extract both running time information and resource access information from existing native or ARINC-653 applications running under the PikeOS operating system from SYSGO.
One of the main requirements of such an environment was to make it as low intrusive as possible in terms of probing time (and resource access) to not bias the collected results. As it was not possible to integrate it directly into the operating system (source code is not available), we had to integrate the probing mechanics into the application themselves. We made a significant effort to minimize the intrusiveness in the application source code, by selecting an ad'hoc software architecture for the tool suite.

2.3.2 METrICS Software Architecture

METrICS consists of several components appearing in Figure 2.4. The brown parts in the figure correspond to, from bottom to the top, the selected hardware architecture; and the PikeOS operating system above the platform support package (PSP) corresponding to the board. The blue parts correspond to the running applications we wish to monitor, and finally the green parts correspond to the components of METrICS further described below.

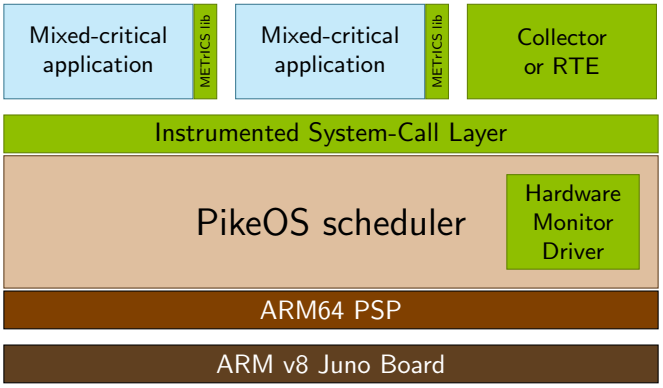

Figure 2.4: Architecture of the METrICS measurement tool

The METrICS measure environment is composed of:

- a **library** to be linked to the applications to provide them with the probing features allowing us to collect time and resource access information.
- a **collector**, implemented as a native PikeOS partition, storing all the collected information and responsible for sending this information to an external host.
- a **kernel driver**, running in privileged mode, allowing us to perform the configuration of the Performance Monitor Counters (PMC) we wish to collect on the board.

- some **post-processing scripts**, extracting understandable information from the collected data, and computing the budgets driving the runtime engine.

2.3.2.1 METrICS Library

The METrICS library is meant to be linked with the running applications to provide them an access to the API corresponding to the measurement probes. The library contains: 1) the instrumented system call layer; 2) the application instrumentation interface; 3) the user-level interface to the instrumentation kernel driver; and 4) the user-level interface to the collector.
syscall instrumentation layer: The instrumentation of the PikeOS system calls (and the APEX system calls in case of ARINC-653) applications automatically inserts measurement probes before and after some system calls. It especially allows us to determine communication times for the intra-partition and inter-partition communications which rely on such system calls, and that can be a significant part of the application running time.
application instrumentation: Besides the syscall-level instrumentation, we provide the ability to manually insert measurement probes directly in applications. This is achieved by adding a pair of functions (`metrics_probe_begin()` and `metrics_probe_end()`) around the section of the code to be monitored. Upon execution, these functions collect the timebase counter corresponding to the number of elapsed cycles since booting and the PMC registers of the current core. The latter function is also responsible for sending the monitored data to the collector.

2.3.2.2 Kernel Driver

On most hardware architectures, the access to hardware performance counters requires supervisor-level privileges. Also on all hardware targets, the configuration of these registers to select the event that should be counted out do require these privileges. As PikeOS legitimately prevents applications from getting such privileges, it was necessary to develop a kernel driver (a new feature of PikeOS 4.0).
The services provided by the developed driver are: 1) selecting the hardware events monitored by the hardware performance counters of local processor core. 2) starting the counters. 3) stopping the counters.
The user interface for these services, implemented in the library, uses the regular ioctl calls provided by PikeOS for drivers. For the Juno board, $\sim$50 events can be selected for 6 different counters.

2.3.2.3 Collector

The collector is a native PikeOS partition, which role is: 1) to define a shared memory space where each instrumented application will save its collected measurements; 2) configure specific measurement scenarios (like selection of events via the driver); 3) launch the measurement campaign (relying on PikeOS schemes); 4) transfer the content of the shared memory to the host computer, either at the end of the measurement campaign or during the run.
The collector is used during the characterization phases used to determine the budget in terms of resource access as explained in deliverable D3.2. During final execution phases this partition is replaced by the run-time engine native partition.

2.3.2.4 SAFURE Budget-Based RunTime Engine (BB-RTE)

The SAFURE Run-Time Engine is a replacement native PikeOS partition aiming at ensuring timing and resource sharing integrity at execution time. It replaces the collector providing the same initialization features (selecting the events and launching the campaign). However the collected data is not sent to another host, but used to adapt the scheduling during the execution.

To guarantee running times for critical applications, the RTE has the ability to suspend less critical applications once they have reached their maximum allocated budget in term of resource access. This BB-RTE is furthermore detailed in the deliverable D3.2.

2.3.3 METrICS internal operation

2.3.3.1 Scheduling Policies

In order to minimize the runtime intrusiveness on application codes, the collector is executed outside of the operational scheduling. We distinguish three scheduling phases in the course of a measurement campaign implemented with ”scheduling schemes” in PikeOS:

- **SCHED_BOOT** during which only PikeOS and the collector are running in time partition 0 (best effort) of PikeOS. During this scheduling phase, the collector performs the initialization of the measurement campaign, preparing the shared memory and the PMC of each core. Once the initialization is completed, the collector shifts to the MONITORING scheme.
- **MONITORING** during which the mixed-critical application partitions are scheduled according to their unmodified deployment scheme, using the collector's shared memory to store their collected metrics. During this scheme, the collector remains blocked on an incoming transmission from the application to notify for the end of the execution, and is therefore not schedulable. Upon receiving this notification, the collector shifts back to the SCHED_BOOT scheme.
- **SCHED_BOOT** during which again only PikeOS and the collector are running. During this phase, the collector performs the transfer of the collected measurements to the host using the MUXA protocol of PikeOS.

Using such a set of schemes allows us to minimize the time intrusiveness of the measurement environment regarding the applications. Indeed, all MUXA communications, system calls to the driver, and the collector itself are not running at the same time as the application. It is therefore not necessary to dedicate a time slot for the collector, and the original schedule of the application is unchanged. When running together with the runtime engine instead of the collector, the scheduling phases remain the same, with the RTE running together with the applications during the MONITORING phase. Also, there is much less data to transmit to the host (mostly watchdog information) during the last phase.

2.3.3.2 Measurement collection format

The collection of the collected metrics is stored in the shared memory initialized by the collector. Each measurement is performed either automatically through the instrumented system call layer, or manually inside the applications as explained above.
The call to the corresponding `metrics_probe_begin()` and `metrics_probe_end()` API automatically store information into this collection memory using the event format presented in Figure 2.5.

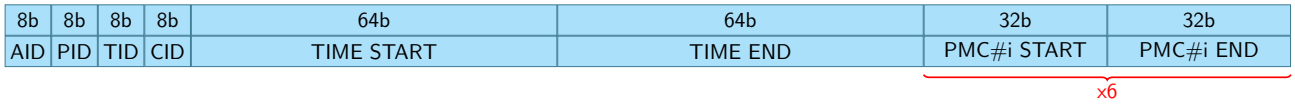

Figure 2.5: METrICS collection format

Each measured event contains current time (in elapsed cycles since booting), current values of hardware performance counters, as well as application, thread, partition and core identifiers. The structure a collected event is the following:

- **AID**: Application identifier.
- **PID**: Partition identifier for multi-partition applications.

- **TID**: Thread identifier for multi-threaded applications.
- **CID**: Core identifier the thread is running on.
- **TIME pair**: Cycle time value upon the call to the `metrics_probe_begin()` and `metrics_probe_end()` functions.
- **PMC#i pairs**: Value of the i^{th} PMC upon the call to the `metrics_probe_begin()` and `metrics_probe_end()` functions.

2.3.3.3 Transmission of collected measurements to the host

At the end of the execution the collector is responsible for transmitting the collected data to the host. This transmission is performed using the MUXA service provided by PikeOS to transfer the content of the collector shared memory.
To avoid the regular issue with transmitting binary data through this channel (endianess issue, control command issues). We transmit this data directly in ascii format as the content of a mon.csv file. The format of this file is strictly the same as the collection format described above.
We dedicated a MUXA channel for this communication (channel 4, port 1506) initially doing MUXA over UART. However we quickly reached the maximum throughput of the UART controller, so we had to switch to MUXA over Ethernet. The implication is that PikeOS has to support the Ethernet controller of the target board, and that one of the Ethernet port has to be dedicated to MUXA (which itself is able to multiplex). This is the case for the ARM Juno board.
On the host side, a driving script is performing a telnet connection to the MUXA client to dump the collected data directly in the corresponding mon.csv file.

2.3.4 METrICS usage

The METrICS environment allows us to collect various metrics during the execution of safety critical applications including running time and shared hardware resource access information. Rather than only extracting worst cases focusing only on average and maximum values, the METrICS tool suite extracts the whole distribution of each measured data.
As a consequence, the amount of gathered data is quite large, with 1.5MB of data being collected for a single run of our test application; 390MB of data being collected for a full run set corresponding to a specific mapping of the applications, and more than 7GB of data being collected so far for the full design space exploration.
To deal with such large evaluation traces, we had to automatize the post processing of the data to extract meaningful information.

2.3.4.1 Post-processing collected information

After a run is executed, the data is collected on the host in the form of a pair of CSV files: **info.csv** that is describing the run (which application were executed, what was the mapping, which level of additional stressing, ...), and **mon.csv** that corresponds to the monitored data during the run, with a file format corresponding to the measurement collection format described in Section 2.3.3.2.
The post-processing phase, realized with a set of Python scripts is in charge of processing these CSV file to extract and visualize meaningful data.

2.3.4.2 Extracting runtime information

From the time integrity point of view of the safety critical systems, the goal is to ensure that all the critical tasks have their deadlines guaranteed. As a consequence, the first kind of information our Python scripts are extracting are the runtime information of the tasks composing the application.

A classical way to visualize this timing data is to build a chronogram that represents the time on the x-axis and the different tasks on the y-axis. Figure 2.6 corresponds to the kind of chronogram METrICS is able to build. It shows for each task (Generator, Splitter, Filter passes, Aggregator and Display) the start / end time of the task as well as its duration.

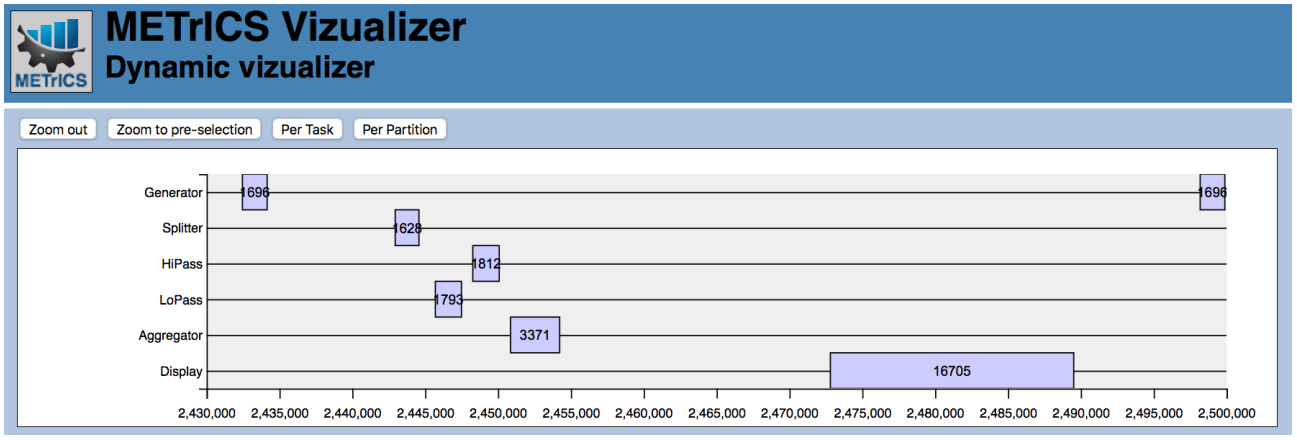

Figure 2.6: Chronogram of the task execution times

Even that such a chronogram view is nice to figure out what is running in parallel at a given time, it does not help to easily identify runtime variation during different iterations of the same task, which is quite important from the worst execution time point of view.
To better observe this runtime variability we also build a runtime histogram for each task, as appearing in Figure 2.7.

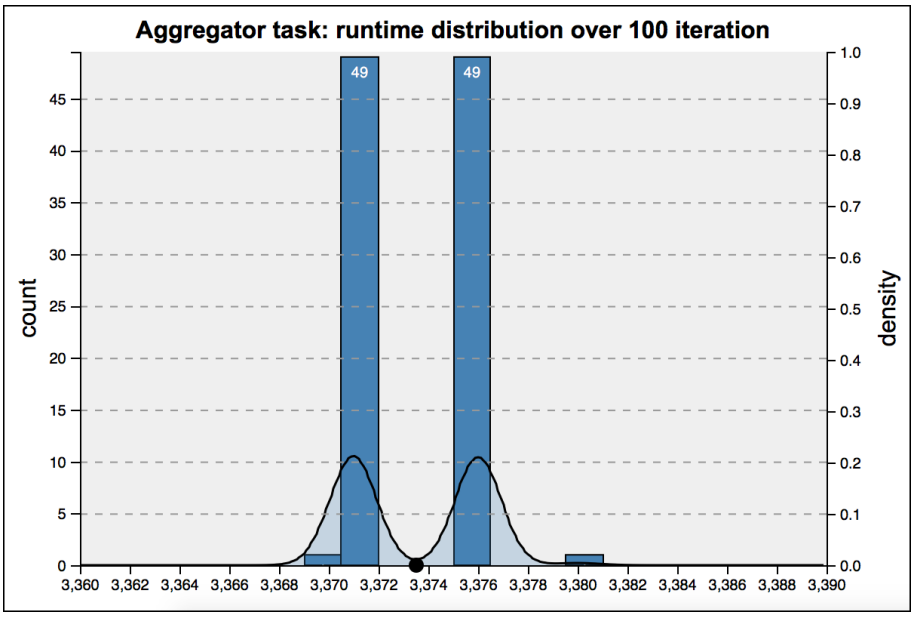

Figure 2.7: Histogram of the task execution time

Such a histogram represents the full distribution of all the iteration runtimes of a particular task. It allows us to identify different time behavior patterns represented by different cluster of values, as well as to determine the representativity of each pattern. Alongside with minimum (leftmost) and maximum (rightmost) values, we also illustrate on the figure with a black circle on the x-axis the median execution time value. Finally the Gaussian curve and the right axis are the associated kernel density estimation.

2.3.4.3 Correlating execution times with hardware resource accesses

Our Python post-processing scripts are able to extract histogram information for hardware resource access as well. By comparing the time distribution of both the resources accesses and the runtime, it allows us to figure out if a particular hardware resource as an impact on the application performance. To further identify such correlations, the Python scripts are also able to extract correlation diagrams, such as the one appearing in Figure 2.8.

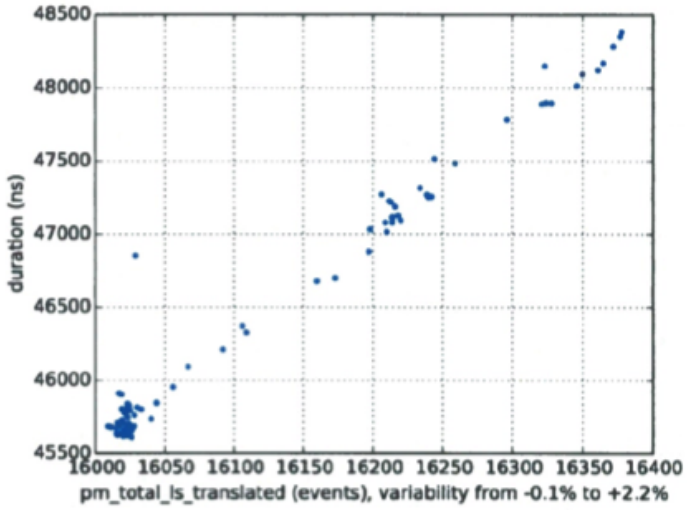

Figure 2.8: Correlation diagram between runtimes and resource accesses

For such a diagram, the y-axis represent the runtime while the x-axis represent the number of accesses to a particular hardware resource. A point in the diagram means that a run exists with such a duration performing this particular number of accesses. If all the points are forming a line such as in Figure 2.8, it means that a linear correlation exists between the runtime and the accesses to this resource. If such a correlation does not exist, the points will form unrelated clusters of data.
This correlation diagram is allowing us, for each application, to figure out to which shared hardware resource the application is the most sensitive to. It will allow us to focus only on these resources while doing runtime monitoring in our RTE.
Another option is to identify correlation among access counters to eliminate redundant information provided by correlated hardware resources (like the redundant information of L1 cache misses versus L2 cache accesses).

2.3.4.4 Budgeting resource access to drive the SAFURE RunTime Engine

Once we have identified for an application which are the resource that correlates in terms of access with the application performance, we can figure out the maximum budget in terms of resource access that will guarantee that the application will be able to run under specific time deadlines.
The Budget-Based RunTime Engine (BB-RTE), developed within SAFURE to provide time integrity while monitoring and budgeting resource accesses is fully described in Deliverable D3.2. The associated results will be presented in Deliverable D4.2.

Chapter 3 Integrity Methodology for Networks

This chapter will describe the methodologies developed in SAFURE for ensuring timing and resource sharing integrity. They are divided in methodology for multi-cores and networks. Both rely on mechanisms described in D3.2.

3.1 Ethernet Timing Analysis Methodology

This section describes a methodology for networks (in particular Ethernet), that ensures proper timing isolation at design time. It employs a model-based worst-case analysis which computes worst-possible timing effects for a given network configuration in order to verify that a certain network configuration meets the timing requirements under all circumstances. By systematically modeling possible error scenarios, the analysis can also verify that timing is met under error conditions (like babbling idiots, i.e. faulty nodes sending unexpectedly).

3.1.1 Network Model

The proposed timing integrity methodology for networks relies on a model-based analysis of the network. The model consists of

- Topology model (switches, links, ECUs/nodes)
- Traffic model (messages, sizes, activation patterns, priorities)
- Constraints (message deadlines, buffer sizes)

The model of topology and traffic can be created manually in SymTA/S or imported from existing descriptions resulting from the network design process using industry standard formats (e.g. DBC, FIBEX, AUTOSAR), see Figure 3.1.

3.1.1.1 Design Rules for Constraint Specification

To avoid time consuming specification of constraints for individual model elements (e.g. deadline for each individual message), they can be generated according to design rules.
Suggested design rules:

- Worst-case load of all switch ports should be below 80%. This allows some headroom for uncertainties and avoids congestions in case of non-constant loads.
- Worst-case load of transmitting ECU ports should be below 50% in normal cases. This avoids single ECUs of spamming the network. This rule can be violated in special situations (e.g. 100Mbit-ECU connecting to Gbit-switch)
- Worst-case latency of periodic messages of the highest priority should be below their period.
- Worst-case latency of periodic messages of the medium priority should be below twice their period.

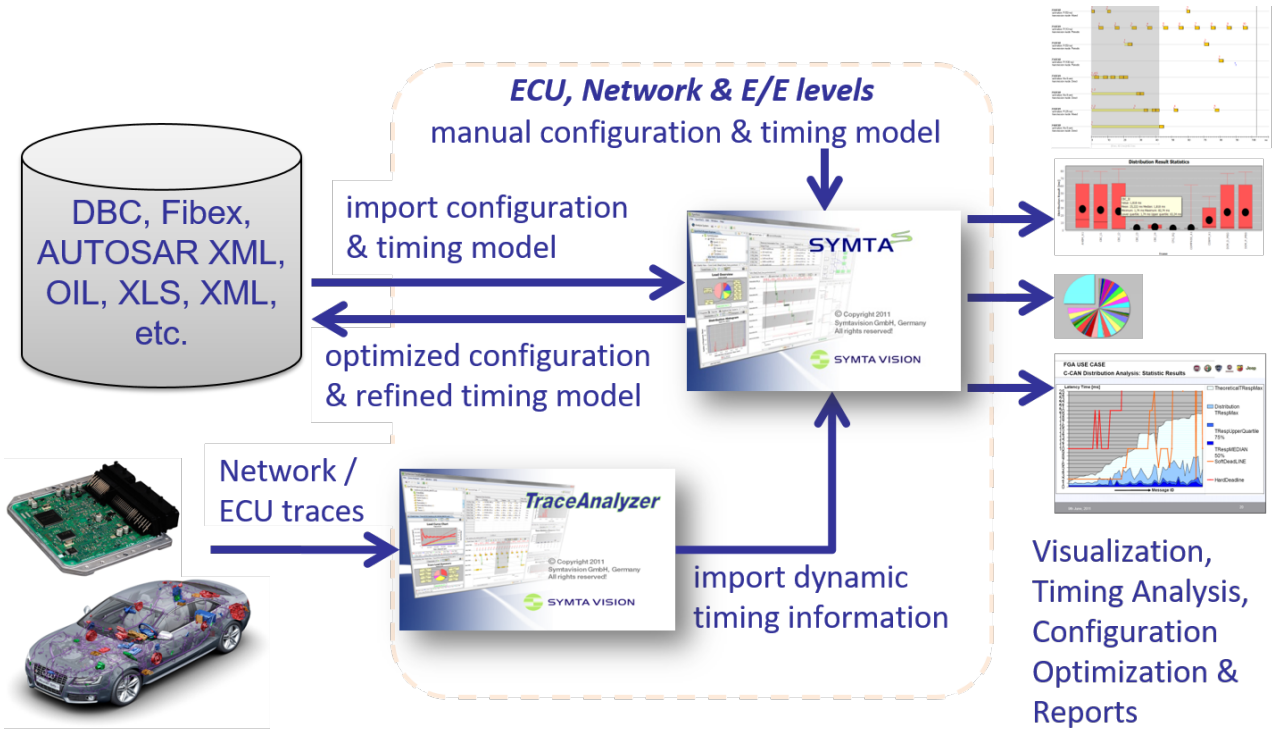

Figure 3.1: Symtavision Tool Suite integrated in automotive developing workflow

- Worst-case latency of periodic messages of the lowest priority should be below ten times their period.
- Worst-case buffer occupancy for all switch ports should be below the physical buffer capacity of the port according to the specifications of the switch. This avoids frame drops for all messages.
- Alternatively (if supported by the switch), the worst-case buffer occupancy can be constrained per priority, according to the priority-partitioning configured in the switch. The constraint should only be specified for priorities transmitting critical messages (where frame-drops are not allowed) or medium-critical messages where no end-to-end protection against frame drops is used (e.g. UDP protocol).
- Worst-case buffer occupancy for all ECU ports should be below the physical buffer capacity of the port according to the available TX memory of the ECU. This avoids frame drops/congestion for all messages.
- Alternatively (if supported by the ECU), the worst-case buffer occupancy can be constrained per priority, according to the priority-partitioning configured in the ECU. The constraint should only be specified for priorities transmitting critical messages (where frame-drops are not allowed) or medium-critical messages where no end-to-end protection against frame drops is used (e.g. UDP protocol).

3.1.2 Analysis of Timing in Error-Free Case

The first step of the analysis methodology is to ensure the timing requirements are met in the error-free case. For this, the worst-case timing analysis algorithm developed in WP3 is used on an unmodified model of the network. It computes upper bounds for the Ethernet port loads, message latencies and switch buffer occupancies that include the worst possible behavior according to the specified system. The computed bounds are compared to specified constraints.

In case all constraints are met, the network timing integrity in the error-free case is guaranteed. If some constraints are violated, the system configuration has to be adapted before evaluating error scenarios. This involves architectural changes and/or changes in the traffic model (which hast to be communicated with function owners to ensure system functionality with reduced traffic requirements) Design guidelines for handling constraint violations:

- Load violation: Optimize topology to avoid bottlenecks; Re-route parts of traffic using static routes (if possible); Reduce traffic by adjusting traffic model (e.g. increase period for periodic messages, if possible); increase transmission speed.
- Latency violation: increase constraint (if tolerable by functionality); reduce frequency of message transfer (if tolerable); increase message priority; optimize topology to reduce distance to receiver and/or reduce bottlenecks; increase transmission speed; implement traffic shaping for same- and higher-priority traffic to reduce long bursts of interference.
- Buffer occupancy violation: Implement traffic shaping to avoid bursts; re-partition buffer allocations; optimize topology; increase transmission speed of outflowing port.

The specific guidelines can be supported by analysis results (e.g. identification of bottlenecks, identification of bursts, ...).

3.1.3 Analysis of Timing in Case of Babbling Idiots

The previous analysis only assured that there are no violations of timing in case all traffic (and components) are behaving as specified. In particular, this means that for instance a low-criticality message is sending only according to the specified message size and activation pattern.
To evaluate the network's integrity against a babbling idiot (e.g. due to a faulty ECU), the babbling traffic needs to be added to the traffic model. This is done by adding messages with very high payload and/or high activation frequency to each ECU that can potentially be a babbling idiot. The priority of these messages should be set to the highest value allowed by the corresponding ECU (assuming priority enforcement is available, e.g. by ingres policing at the switches).
Running the analysis on the modified system will show how critical messages behave under the presence of a babbling idiot. In case no constraint violations (of critical messages) are present, the system is robust against this type of error. If violations occur, they must be addressed by the guidelines above or (possibly in addition) by implementing more/better timing isolation mechanisms (e.g. traffic shaping implemented according to high criticality, switch-level traffic policing).

3.2 Network Integrity

In this section we introduce an approach to setup a network communication channel in PikeOS with a configuration of shared resources to mitigate impact of timing covered channels. The channel uses a Virtual Private Network to provide integrity and confidentiality of communication. The resource separation of a running PikeOS system minimizes timing covert channels.

3.2.1 VPN

A Virtual Private Network (VPN) is a type of connection which allows the setup of a private connection between two remote links over a transit network like e.g. the Internet. After the connection is set up, the links can interact with each other like they have a wired connection to a local network.
The basic topologies are:

- Host-Host
- Host-Network

- Network-Network

A private network is used to provide a group of actors an access to data or internal communication facilities which can be sensitive to leaks or corruptions. Also network attacks like “Man in the Middle” attack or “Packet injection Attack” can be mitigated by using a private network with data integrity validation, data encryption, or both. To prevent an unauthorized access and also to ensure integrity and confidentiality of the communication, the private network has to provide at least all of the functionality listed below:

- User Authentication.
- Address Management.
- Data Encryption.
- Key Management.
- Multi-protocol Support.

3.2.1.1 VPN with IPSec

Internet Protocol Security (IPsec) is a protocol suite for secure Internet Protocol (IP) communications by authenticating and encrypting each IP packet of a communication session. IPsec includes protocols for establishing mutual authentication between agents at the beginning of the session and negotiation of cryptographic keys to be used during the session. IPsec can be used for protecting data flows between a pair of hosts (host-to-host), between a pair of security gateways (network-to-network), or between a security gateway and a host (network-to-host). Internet Protocol security (IPsec) uses cryptographic security services to protect communications over Internet Protocol (IP) networks. IPsec supports network-level peer authentication, data origin authentication, data integrity, data confidentiality (encryption), and replay protection. IPsec is an end-to-end security scheme operating in the Internet Layer of the Internet Protocol Suite, while some other Internet security systems in widespread use, such as Transport Layer Security (TLS) and Secure Shell (SSH), operate in the upper layers at the Transport Layer (TLS) and the Application layer (SSH). Hence, only IPSec protects all application traffic over an IP network. Applications can be automatically secured by IPSec at the IP layer [33]. The following sources [16] [10] [29] give a good presentation of IPSec.

3.2.1.2 VPN in PikeOS

IPSec works on the network layer and it is a part of a network stack. PikeOS provides partitions access to a physical network interface. It means that different network stack implementations can work in different partitions simultaneously. Also it is possible for partition with a network stack to provide communication facilities to other partitions using the PikeOS file provider API.

The PikeOS hypervisor has a microkernel architecture and comprises external components to provide additional functionality. The following network stack components are available for PikeOS:

- CIP
  A certified UDP/IP network stack compatible with the standard RFC specification.
  It is implemented as a PikeOS file provider. It operates in a separate partition and can serve more than one partition at a time providing a UDP socket API. It was developed to be a certifiable and thus it provides only minimal required functionality.
- PikeOS port of LwIP
  LwIP is an open source TCP/IP stack designed for embedded systems [34].
  The PikeOS LwIP port supports the following network protocols: IP, ARP, ICMP, RAW, UDP,

DNS, DHCP, TCP and socket API. PikeOS provides a LwIP port as a precompiled library which depends on a PikeOS implementation of the POSIX specification. Both the LwIP and PISIX libraries have to be included in an application build process. Since the LwIP port is a library which is a part of a PikeOS application, it can not be scaled i.e. one instance of the LwIP port can not be shared between applications in different PikeOS partitions.

- PikeOS port of NetBSD network stack
  This port provides socket interface over TCP, UDP and IP. It uses NetBSD network subsystem with replaced NetBSD system calls with PikeOS calls. The current implementation is designed to be working in a single PikeOS partition.

ANIS and LwIP provide basic network functionality and IPSec is beyond their typical use case. Implementation and integration of IPSec in any of them from scratch is a time and recourse consuming task.
The PikeOS port of NetBSD network stack is based on an old version of NetBSD what provides an additional overhead. It needs to be updated first and only then extended by porting missing components like crypto engine and Internet Key Exchange (IKE) protocol.

To fit in the time and resource budget of the SAFURE project, the decision was made in a favor of adaptation of an existing network stack with IPSec and IKE support.
The result of a research of network solutions available on the market was the following two candidates:

- 6WIND [1]
  This network stack with IPSec is focused on hi speed servers.
- UNICOI [30]
  This network stack with IPSec is focused on embedded systems.

The solution from UNICOI was considered as the most suitable one.
Table 3.1 summarizes the results described above.

	UNICOI FNS	**CIP**	**LwIP port**	**NetBSD port**
Robustness	Stable codebase	Certified	Stable codebase	Stable codebase
Portability	Portable core	Native	Requires POSIX	Requires internal modifications
Socket API	Yes	Only over IP4 UDP	Yes	Yes
SLOCC	n.a.[1]	8671	44393	n.a.[1]
Supported Security	IPSec, IKE	no	no	Requires internal modifications

Table 3.1: Network stacks comparison table

Architecture

The PikeOS VPN solution uses a similar approach as the PikeOS ANIS network stack. It is implemented as PikeOS file provider which executes in a separate partition and provides a socket API to other partitions. It allows to have a single connection point to a private network which can be used by

[1] expected to be more more than CIP or LwIP

a number of partitions configured to have access to the connection. This simplifies configuration of the system, and at the same time, it leaves flexibility to have a number of VPN partitions connected to different private networks, if such a configuration is required.

The architecture of the PikeOS VPN file provider is shown in the Figure 3.2.

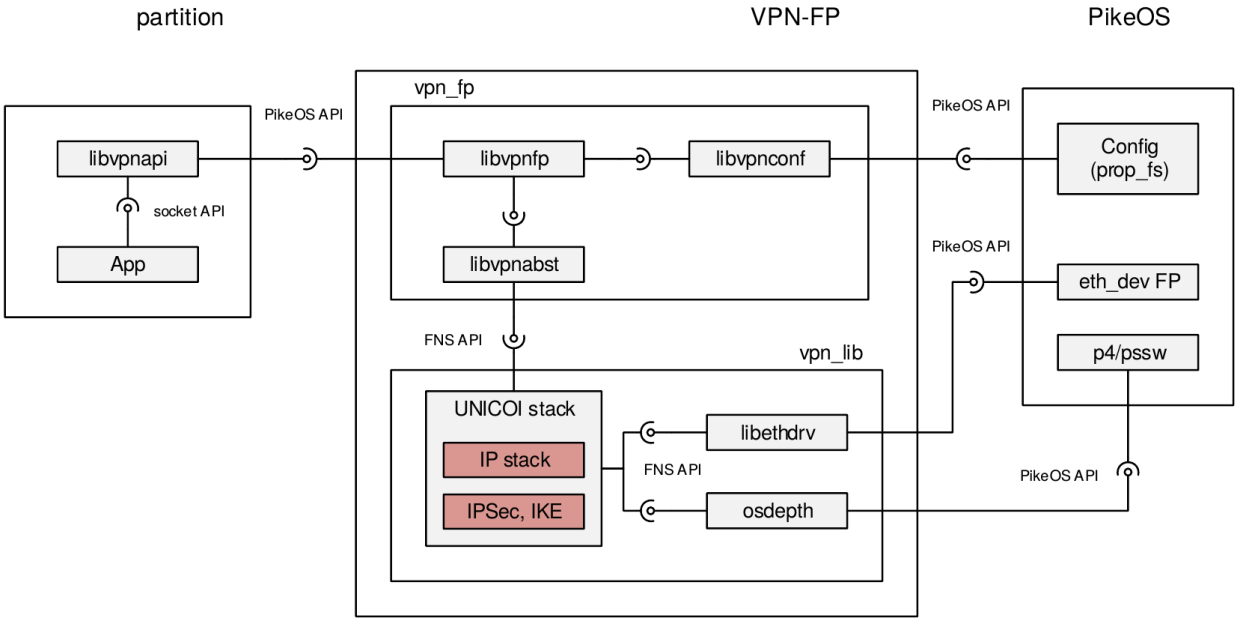

Figure 3.2: PikeOS VPN File Provider

The PikeOS VPN consists of the following components:

- **vpn_fp**
  This is a high level component of the PikeOS VPN module.
  - **libvpnfp** is responsible for multi user operation. Together with **libvpnapi** it provides abstraction of the PikeOS file provider API over socket API.
  - **libvpnconf** provides configuration of the module by employing PikeOS.
  - **libvpnabstr** is an abstract layer between general VPN functionality and third party implementation.
- **vpn_lib**
  This component is a the third party network stack and VPN implementation, integrated into PikeOS.
  - **IP stack, IPSec, IKE** are OS-independent parts of UNICOI network IP stack, IPSec and IKE, which provide API for configuration and management of network subsystem.
  - **libethdrv, osdepth** contain the OS/hardware dependent parts of the IP stack, IPSec and IKE.

3.2.1.3 Use Case

A use case of PikeOS VPN file provider is shown at Figure 3.3.
There are three partitions: two application partitions and one VPN file provider. PikeOS provides means for logical separation of a physical network device at a driver level. App1 and VPN-FP can use the network device simultaneously. To be able to send packets over the network App2 has to have its own network stack implemented within the application.

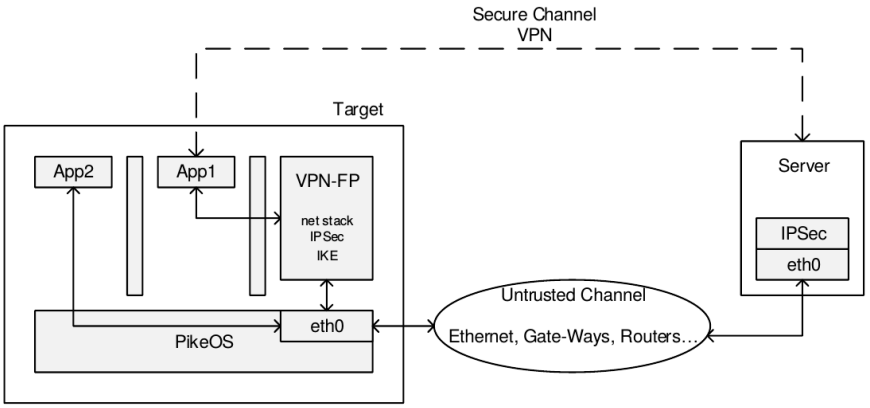

Figure 3.3: vpn_use-case

On the other hand, being properly configured by a system integrator the VPN-FP can serve a number of PikeOS partitions providing them a widely used socket API. Thus App1 can communicate with a server or remote device inside the private network.
The VPN-FP can be configured to provide partitions access to both public and private networks or to restrict access to the private network only. Such restriction reduces the attack vector on the communication channel since transfered data is protected by authentication, integrity and cryptographic algorithms.

Using the PikeOS VPN file provider allows application developers to reduce the complexity of an implementation and setting up a safe and secure communication channel for their applications; and focus on the functionality.

3.2.2 Time and Resource Partitioning

Figure 3.4 shows example of clustering armv8 cores and the resulting cache coherency groups. There are two clusters containing two cores each. Each core in a cluster has a dedicated L1 cache, while an L2 cache is shared within the cluster. All clusters in SoC shared the interconnect.

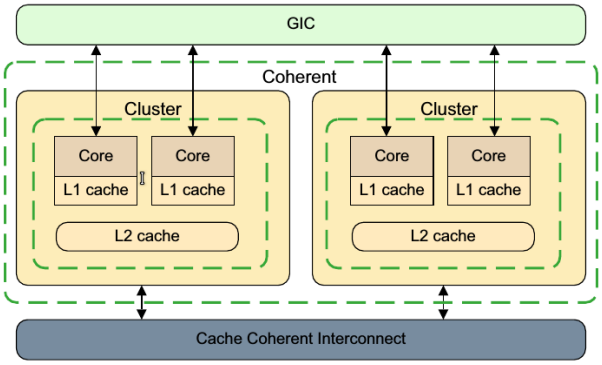

Figure 3.4: Cache coherency groups

3.2.2.1 Threats

A concurrency on interconnect and L2 shared caches can be leveraged by an attacker to perform one of following attacks.

Interconnect Covert Channel

When mixed criticality applications are integrated in multicore platforms such as Juno Board, the applications can be mapped to the cores in multiple ways. Thus applications assigned to different cores can still share some internal resources. If the system is configured without interconnect coherencies in mind then the cores issuing concurrent memory transactions might be served by the same memory controller through the shared interconnect.
A sharing of an interconnect and memory controller introduces interferences between concurrent partitions running from different cores and this interference can be leveraged to form a timing covert channel between the partitions. Wu et al. [35] describe such a covert channel based on the contention on the memory bus on Intel platforms. In [35], the bus contention is created by issuing atomic memory instructions that lock the bus during their operation.

L2 Cache Covert Channel

In the deliverable D3.1, we introduced a study [17] where two partitions leverage the L2 shared cache to setup a covert channel to exchange data between them.

Other L2 Caches Attacks

The covert channel mentioned in the paragraph above assumes that Transmitter and Receiver are co-operating with each other for information exchange. There have been many studies on non co-operative covert channels using a shared L2 cache to extract secret information of a cryptographic operation by observing the cache activity of the victim [38, 22, 32]. Among these, the two approaches used by Osvik et al. [22] are called “Evict+Time” and “Prime+Probe”.
In the **“Evict+Time”** method, the attacker evicts the cache lines of victim and then triggers an encryption operation. The secret key is extracted by observing the time taken by the victim to perform the encryption. This attack uses the variation in encryption time based on whether the lookup table lines are cached or not.
In the **“Prime+Probe”** method, the attacker loads the cache with its memory block and triggers the encryption operation at the victim. After the encryption operation, the attacker accesses the memory block again and based on the access time, it can guess the lines of lookup table used by the victim during the encryption. The approach we used in our implementation is the co-operative version of “Prime+Probe” method.

In 2014 Yarom and Falkner introduced **"Flash+Reload"**. It is a cache side-channel attack technique that exploits the weakness to monitor access to memory lines in shared pages [37]. It allows to determine, with a very high accuracy, which part of a library or a binary was accessed be a victim. This attack can also be used against encryption algorithms to extract components of private key.

A good example of how applications of different types of attacks on L2 cache on the ARM mobile platform architecture can bring severe consequences for end users was demonstrated by Moritz Lipp & Clmentine Maurice at "black hat" conference in 2016[18]. As an example they introduced an Android keylogger that was able to determine which type of key is pressed e.g. character, backspace, enter key etc. The keylogger did not require any permissions and did not interact with any application directly.

3.2.2.2 Mitigating the Threats

To mitigate threats listed above the following actions can be taken:

Resource Partitioning

In platforms like the Juno Board, two memory controllers are available (see Figure:3.5). In such platforms, a memory controller can be dedicated to serve the memory accesses from one partition.

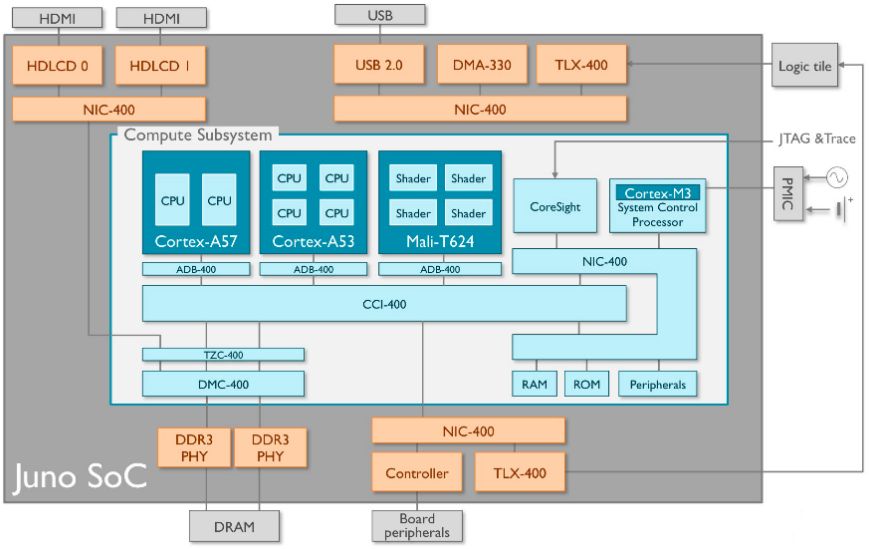

Figure 3.5: ARM Juno SoC architecture

The PikeOS configuration allows to assign a partition a particular core. By partitioning the memory controller between concurrent partitions, the timing covert channel between them can be minimized. Since there are two memory controllers available on the Juno platform, two partitions can be formed in this way, as shown in Figure 3.6.

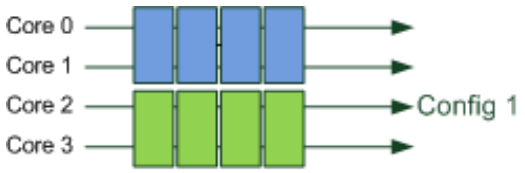

Figure 3.6: Example of a resource partitioned configuration

This configuration uses less CPU power in favor of more responsiveness of the system. It mitigates timing covert channels while leaving a L2 cache attack possible.

Time Partitioning

The other approach for mitigating timing covert channels is to separate execution of concurrent partitions in time. PikeOS time partitioning allows such separation.

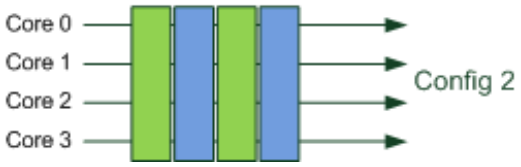

Figure 3.7: Example of a time partitioned configuration

Figure 3.7 shows the configuration where two partitions are running in separate time frames. It allows usage of all available cores by each partition for a fixed period of time. Like the example above this approach mitigates timing covert channels and leaves L2 cache attack possible.

Cache Flushing and Sandwich Partition

Cache based timing covert channels can be completely eliminated by introducing a buffer partition between the partitions having information flow restrictions. The width of this partition shall be more than the worst case execution time to flush the data cache and TLB and invalidate the instruction cache. By inserting such a *sandwich time partition*, any timing variation on cache flushing operation based on the cache state will not be observable from the next partition.

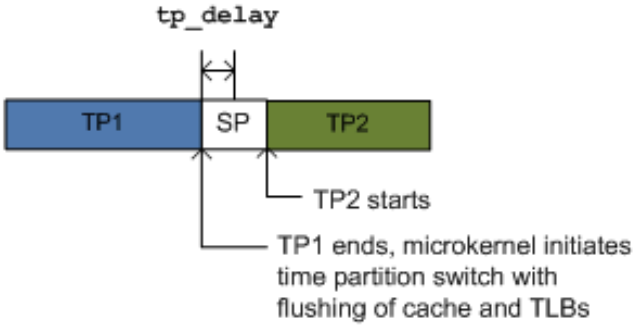

Figure 3.8: Example of time partitioning with a sandwich partition

Figure 3.8 shows a time partition scheme with *sandwich partition (SP)* inserted between two critical applications running from time partitions TP1 and TP2. By inserting the SP, *tp_delay* is contained within the SP and the execution time of TP2 is not affected by the cache/TLB flushing.

This approach is an extension of the timing separation introduced above. It results in time overhead caused by a sandwich partition, but at the same time, it is effective against both timing and L2 types of attacks.

3.2.2.3 Combined Solution

As shown in this chapter, only software implemented protection of the entire system may not be sufficient. Studies [22, 37, 18] show that even well implemented software still may remain vulnerable because of the hardware where it is executing.

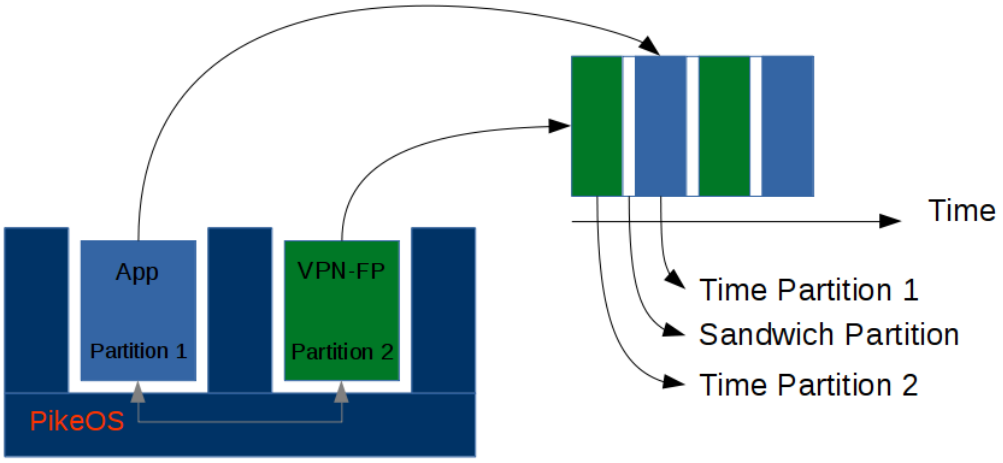

Figure 3.9: Example of time and resource partitioning

Figure 3.9 shows a PikeOS project configuration where VPN-FP solution is combined with time partitioning. Here the VPN implementation is separated from the application using it. This provides integrity on the network communication level. At the same time a time partitioning between application using VPN connection and the VPN implementation mitigates attacks on cryptographic algorithms and also mitigates covert channels introduced above.
A guidance on how to configure a PikeOS project in this way will be introduced in SAFURE delivery D6.7 ”Final specifications of the SAFURE Framework and Methodology”.

Chapter 4 Summary and Conclusion

This document reported the integrity methodology developed in SAFURE. It presented the various guidelines and tools that form this methodology and specifically showed how tools and mechanisms developed in this project can be used together to ensure the safety and security of embedded systems.
For multi-cores, Chapter 2 presented a temperature integrity methodology, as well as guidelines for vulnerability detection and a measuring environment for time-critical systems.
In the thermal protection research thread, Section 2.1.2 demonstrated a scheme which can provide thermal protection and thermal isolation in a mixed-critical system. The performance of the proposed scheme is highly dependent on the accuracy of the thermal model. Section 2.1.3 shows that temperature and operating frequency mediums pose a significant security risk. Covert channels based on these mediums can achieve considerable throughput (45 bps and 0.9 bps for thermal covert channel and frequency covert channel, respectively). Therefore, mitigation strategies need to be in place to counter the security risk. Section 2.1.3 also highlights possible mitigation strategies.
Section 2.2 introduced the strategy to use the signature/template approach for multicore vulnerability detection, described in D3.2. In particular, such strategy includes identifying onchip shared resources, relevant events for those resources, potential contention for those events and finally applying signatures/templates to account for potential contention in the Worst-Case Execution Time Estimates.
A network integrity methodology was presented in Chapter 3, where a formal analysis for verifying isolation properties of Ethernet networks was shown as well as a methodology that minimizes timing covert channels for the secure the communication channel.
The formal analysis methodology for Ethernet showed that it is possible to give formal timing guarantees for communication over Ethernet for certain priority levels, given that the communication behavior of traffic of same or equal priority is known at design time. For these guarantees, it is important that the same- and higher-priority traffic behavior stays as specified. This can be ensured by policing mechanisms, which should be employed for additional safety. The analysis also showed that unconstrained erraneous behavior on lower priority levels is tolerable if it is considered during the analysis phase. Also here, certain policing helps, e.g. to constrain the max. size of Ethernet frames to avoid long blocking times (due to the non-preemptive nature of Ethernet). An alternative would be preemption support, as proposed in the IEEE 802.1Qbu as part of Ethernet Time-Sensitive Networks (TSN).
The configuration of the PikeOS introduced in this chapter provides an additional layer of integrity and also confidentiality of the communication channel. VPN serves data exchange via a communication channel between the target and a remote peer. At the same time the timing separation mitigates leaks of this data from the SoC. Such combination reduces vector of attacks on the target.

4.1 Integration Plan

Table 4.1 presents the integration plan for technologies presented in this deliverable. *Telecom* denotes the telecommunication use-case based on Sony Xperia platform, *Automotive* denotes the automotive use-case based on Infineon Aurix platform, and *Juno* denotes the WP4 use-case based on ARM Juno Board platform. Table 4.1 has been composed and added to this deliverable in response

to reviewer's recommendation. However, please note that this table presents a tentative plan. Final integration details will be presented in SAFURE Deliverable D6.7.

Technology	**Section**	**Use-case**	**Comments on integration**
Thermal protection	2.1.1.1, 2.1.2	*Juno*, *Telecom*	Feasibility and extent of integration on *Telecom* will be assessed in September 2017.
Multicores vulnerability due to contention	2.2	*Juno* *Automotive*	Currently being integrated on the *WP4 prototype* Currently being integrated on the *Automotive* multicore use case
METrICS measure environment	2.3	*Juno* *Telecom*	Currently running on the *WP4 prototype*. Feasibility and extent of integration on *Telecom* will be assessed in September 2017. METrICS requires PikeOS with support for privileged mode drivers and MUXA-over-ethernet.
Ethernet Timing Analysis Methodology	3.1	*Automotive*	Methodology for algorithms presented in D3.1. Implementation in the model-based SymTA/S timing analysis tool to be applied to automotive scenarios.
Network Integrity	3.2	*Telecom*	Feasibility and extent of integration on *Telecom* will be assessed in September 2017

Table 4.1: Integration plan for technologies presented in deliverable D3.3

Chapter 5 List of Abbreviations

AES	Advanced Encryption Standard
API	Application Programming Interface
BB-RTE	Budget-Based Run-Time Engine
CSV	Comma-Separated Values text file storing tabular data
EC	European Commission
ECC	Elliptic Curve Cryptography
EDF	Earliest Deadline First
KMAC	Keccak Message Authentication Code
MAC	Message Authentication Code
METrICS	Measure Environment for Multi-Core Time Critical Systems
PMC	Performance Monitor Counter
PSP	Platform Support Package
RTE	Run-Time Engine
TIS	Thermal Isolation Server

Bibliography

[1] 6WIND. 6wind turbo ipsec. http://www.6wind.com/products/6wind-turbo-ipsec/.

[2] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *International Conference on Security and Privacy in Communication Systems*, pages 86–103. Springer, 2013.

[3] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, 2013.

[4] Elaine Barker. Recommendation for key management Part 1: General (Revision 4). *NIST special publication*, 800(57):1–147, 2016.

[5] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 122–132. ACM, 2012.

[6] Jie Chen and Guru Venkataramani. CC-Hunter: Uncovering covert timing channels on shared processor hardware. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 216–228. IEEE, 2014.

[7] Sung Woo Chung and Kevin Skadron. Using on-chip event counters for high-resolution, real-time temperature measurement. In *Thermal and Thermomechanical Phenomena in Electronics Systems, 2006. ITHERM'06. The Tenth Intersociety Conference on*, pages 114–120. IEEE, 2006.

[8] G. Fernandez, J. Jalle, J. Abella, E. Quiones, T. Vardanega, and F. J. Cazorla. Computing safe contention bounds for multicore resources with round-robin and fifo arbitration. *IEEE Transactions on Computers*, 66(4):586–600, April 2017.

[9] FreeScale. *P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual. Rev 1*, 2012.

[10] Steve Friedl. An illustrated guide to ipsec. http://www.unixwiz.net/techtips/iguide-ipsec.html, 2005.

[11] Luotao Fu and Schwebel Robert. Preempt-rt patch. `https://rt.wiki.kernel.org/index.php/RT$_$PREEMPT$_$HOWTO`. Accessed: 2017-04-05.

[12] Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, pages 1–15. IEEE, 2013.

[13] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP@ ISCA*, page 11, 2013.

[14] Infineon. *XMC4500 Microcontroller Series for Industrial Applications Reference Manual. Rev 1*, 2012.

[15] Mikhail Kazdagli, Vijay Janapa Reddi, and Mohit Tiwari. Quantifying and improving the efficiency of hardware-based mobile malware detectors. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[16] Oleg Kolesnikov and Brian Hatch. *Building Linux virtual private networks (VPNs)*. Sams Publishing, 2002.

[17] Don Kuzhiyelil and Sergey Tverdyshev. Timing Covert Channel Analysis on Partitioned Systems. *In Proceedings of escar Europe, Hamburg 18-19 No2014*, 2014.

[18] Moritz Lipp and Clementine Maurice. Armageddon: How your smartphone cpu breaks software-level security and privacy. https://www.youtube.com/watch?v=9KsnFWejpQg.

[19] Francisco Javier Mesa-Martinez, Joseph Nayfach-Battilana, and Jose Renau. Power model validation through thermal measurements. *ACM SIGARCH Computer Architecture News*, 35(2):302–311, 2007.

[20] Ira S Moskowitz and Myong H Kang. Covert channels-here to stay? In *Computer Assurance, 1994. COMPASS'94 Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security. Proceedings of the Ninth Annual Conference on*, pages 235–243. IEEE, 1994.

[21] J. Nowotsch, M. Paulitsch, D. Bhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 109–118, July 2014.

[22] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, pages 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.

[23] Meltem Ozsoy, Caleb Donovick, Iakov Gorelik, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Malware-aware processors: A framework for efficient online malware detection. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 651–661. IEEE, 2015.

[24] Naser Peiravian and Xingquan Zhu. Machine learning for android malware detection using permission and api calls. In *Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on*, pages 300–305. IEEE, 2013.

[25] Devendra Rai and Lothar Thiele. A calibration based thermal modeling technique for complex multicore systems. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 1138–1143. EDA Consortium, 2015.

[26] Devendra Rai, Hoeseok Yang, Iuliana Bacivarov, and Lothar Thiele. Power agnostic technique for efficient temperature estimation of multicore embedded systems. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, pages 61–70. ACM, 2012.

[27] Ali Shafiee, Akhila Gundu, Manjunath Shevgoor, Rajeev Balasubramonian, and Mohit Tiwari. Avoiding information leakage in the memory controller with fixed service policies. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 89–101. ACM, 2015.

[28] L. Sigrist, G. Giannopoulou, P. Huang, A. Gomez, and L. Thiele. Mixed-criticality runtime mechanisms and evaluation on multicores. In *RTAS*, pages 194–206, 2015.

[29] William Stallings et al. IP security. *The Internet Protocol Journal*, 3(1):11–26, 2000.

[30] Unicoi Systems. Fusion embedded ipsec. http://www.unicoi.com/fusion_secure/fusion_ipsec.htm.

[31] U.S. Department of Defense. *DOD Trusted Computer System Evaluation Criteria "The Orange Book" [DOD 5200.28]*. 1985.

[32] Zhenghong Wang and Ruby B Lee. Covert and side channels due to processor architecture. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 473–482. IEEE, 2006.

[33] Wikipedia. Ipsec. https://en.wikipedia.org/wiki/IPsec.

[34] Wikipedia. Lwip. https://en.wikipedia.org/wiki/LwIP.

[35] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security symposium*, pages 159–173, 2012.

[36] Mengjia Yan, Yasser Shalabi, and Josep Torrellas. ReplayConfusion: Detecting cache-based covert channel attacks using record and replay. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–14. IEEE, 2016.

[37] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security*, volume 2014, pages 719–732, 2014.

[38] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316. ACM, 2012.