SAFURE

D3.1
Interim Analysis of Integrity Algorithms

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-D3.1 / Final 1.00
Work Package:	WP 3
Due Date:	30.4.2016
Actual Submission Date:	29.4.2016
Responsible Organisation:	ETHZ
Editor:	Rehan Ahmed
Dissemination Level:	Public
Revision:	Final 1.00
Abstract:	This document will overview existing thermal, data and timing integrity algorithms. Furthermore, it will cover first results regarding the extension of these methods to safe and secure systems
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

Rehan Ahmed (ETHZ)

Contributors (ordered according to beneficiary numbers)

Christina Petschnigg, Martin Deutschmann (TEC)
André Osterhues, Lena Steden (ESCR)
Mikalai Krasikau (SYSG)
Jonas Diemer (SYM)
Sylvain Girbal (TRT)
Daniel Thiele (TUBS)
Gabriel Fernandez, Jaume Abella, Francisco J. Cazorla, Robin Hofmann (BSC)
Marco Di Natale (SSSA)
Rehan Ahmed, Philipp Miedl, Lothar Thiele (ETHZ)

Disclaimer

Executive Summary

There has been a tremendous improvement in performance and efficiency of processing platforms in the past four decades. System designers have exploited various architecture and device level techniques to bring about this improvement. While the average case performance of these devices has improved, the worst-case performance has degraded.
While the average case performance has increased tremendously, . There is a large gap between the requirements of real-time applications and what architectures of embedded processors offer today. On the one hand, real-time applications need predictability in order to enable safe operation based on worst-case execution time analysis. On the other hand, following the end of Dennard scaling, embedded processors increasingly feature a multicore architecture with shared resources (e.g., last-level cache, memory controller) in order to keep improving performance and efficiency.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

The focus of Work Package 3 is to study algorithms for ensuring integrity of Safe and Secure systems which are the focus of SAFURE project. This document overviews the integrity algorithms and presents first results on the application of these algorithms and/or development of new algorithms for preserving system integrity.

1.1 Meaning of Integrity in the Context of SAFURE

Before delving into the different aspects of integrity, it is important to define the integrity of a computer system. In one of the seminal works titled “Integrity Consideration for computer Systems” [**?**], K.J. Biba states that ”We consider a subsystem to have the property of integrity if it can be trusted to adhere to a well defined code of behavior”. This “code of behaviors” is the specification of a given system.
Deliverable D1.3 states the SAFURE framework specification in detail. However, important aspects of this specification are outlined here to motivate the different aspects of system integrity. As stated in D1.3, SAFURE adopts a unified presentation of properties that make a system dependable. Specifically, in SAFURE we focus on the following system attributes:

- Safety attributes: maintainability, reliability and safety
- Security (as of IT security) attributes : availability, confidentiality and integrity

For ensuring that a mixed critical system has these attributes, we have identified three separate areas of system integrity:

- Temperature integrity
- Data integrity
- Timing and resource sharing integrity

We will now briefly overview these integrity aspects. Details on each of these aspects are given in Chapters 2, 3 and 4.

1.2 Temperature Integrity

Temperature integrity refers to maintaining system temperature below a safe operating threshold. It mainly affects reliability, safety, availability and confidentiality attributes of a safe and secure system. This aspect has become important due to rapid increase in power density of modern processing platforms. High temperature conditions adversely affect the reliability/safety of a system. Since reliability and safety are fundamental requirements of mission-critical real-time systems, adherence to thermal constraints is vital for maintaining system integrity. In SAFURE we study the thermal impact of executing tasks of multiple criticalities on a multi-core platform.
We also identify that temperature can be used to compromise the security of a system by its use as a covert communication channel. This compromises the confidentiality attribute. We study analysis/mitigation strategies for countering these threats to system integrity.

1.3 Data Integrity

Data integrity refers to assuring and maintaining the accuracy of data. Thus it can be said that data integrity techniques aim at preventing unintentional changes to information. Data integrity mainly affects reliability, safety, confidentiality and integrity attributes of a safe and secure system.
Problems in that domain comprise unintended changes to data due to storage, retrieval or processing operations. This also includes targeted changes, unexpected hardware failures, human errors and malicious attackers. Measures to preserve integrity of data are diverse, including application of checksums, error correcting codes as well as cryptographic message authentication codes (MACs) and access control techniques, the latter two of which are considered in SAFURE.

1.4 Timing and Resource Sharing Integrity

Timing integrity refers to the property of a (real-time) system to meet its timing requirements, e.g. deliver a response to an external stimulus in time. Timing integrity mainly effects maintainability, reliability, safety and availability attributes of a safe and secure system.
There are varying degrees of criticality w.r.t. timing integrity, ranging from ”best effort” (e.g. providing some service at all) all the way up to safety-critical control loops with short deadlines (e.g. advanced driver assistance, autonomous driving).
Guaranteeing timing integrity requires that all effects that impact the timing of a certain function are controlled. Typically, this means controlling interference during the sharing of resources (e.g. processor time, memory access, ...). This can be done by providing corresponding hardware/software mechanisms on the execution platform. Giving a guarantee on timing integrity typically involves performing a formal analysis of the timing properties.

1.5 Cross-Domain Concerns in the Design for Integrity

We will also study several cross-domain integrity concerns as part of SAFURE. For instance, several methods for reducing system temperature (maintaining temperature integrity) rely on performance throttling strategies such as DVFS; which may compromise timing integrity of a system.
As another example, many of the existing mechanisms for guaranteeing integrity (of data or safe operation of systems) rely on redundancy in the computations or the data. The need to send redundant information or to perform additional computations needs to be addressed in the timing analysis, creating cross dependencies between the data integrity and traditional safety analysis and the time analysis.
These cross dependencies can be targeted at the time of the analysis, but are better addressed by a design process that tries to synthesize a solution that addresses time concerns at the same time it also maintains other integrity aspects.

Chapter 2 Temperature Integrity

This chapter focuses on temperature integrity measures. It overviews existing approaches for temperature analysis and for mitigating high temperature conditions. This chapter also identifies that temperature can also pose a security risk; by being used as a covert channel to leak information.

2.1 Introduction

2.1.1 Emergence of Temperature Constraints

Over the past three decades, the power densities of processing platforms have risen rapidly (an increase of 32x since 80386 processor). This thread is primarily due to the breakdown of Dennard Scaling (**?**]). The overall trend in processor power densities is given in Figure 2.1. Furthermore, the localized power densities can be orders of magnitude higher. Due to this effect, switching too many transistors at a time generates more heat than can be dissipated, possibly damaging the chip due to exceeding the maximum safe temperature. All modern processing platforms have strict thermal constraints; which have to be adhered for safe operation. For instance, Intel Sandy Bridge processors have a thermal constraint of 100°C **?**]. If processor temperature exceeds this, an emergency shutdown is activated.

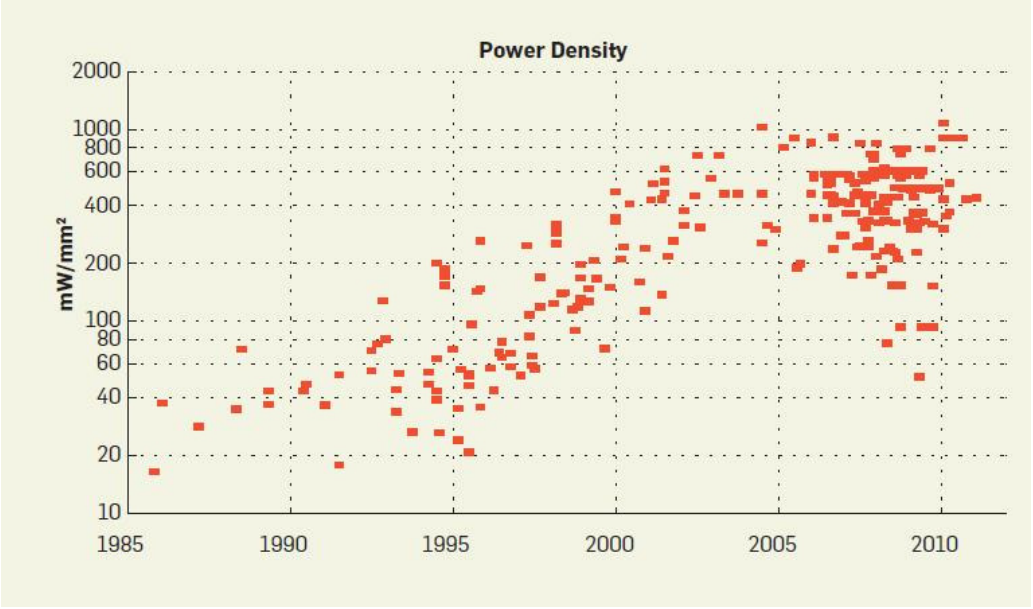


Figure 2.1: Evolution of processor power densities. Figure taken from **?**]

2.1.2 Safety Critical Context

In the general purpose computing domain, high temperature conditions are handled by various performance throttling schemes (DVFS, Dynamic Power Management (DPM) etc). However, such reactive

schemes cannot be directly applied in safety critical applications. This is because, in safety/time critical domains, applications have stringent real-time constraints. Not meeting the time constraints can have catastrophic consequences; including loss of human life. Reactive performance throttling schemes, may lead to a deadline miss and system failure.

2.1.3 Security Context

Temperature sensors are a valuable asset for thermal management, but they can represent a security breach in privilege-separated, or sandboxed, systems. Temperature sensors may be used to implement a covert channel that allows otherwise isolated applications to communicate and possibly leak sensitive data; compromising system security. These sensors are used in all modern processing platforms to mitigate the performance loss due to high temperature conditions. While hardware Dynamic Thermal Management (DTM) can avoid damages and ensure integrity, it resorts to techniques (e.g., sharp speed throttling) that severely impair performance. For this reason, temperature sensors are made accessible via software, to expose data for smarter thermal management policies that gracefully impact performance and avoid triggering hardware DTM. For example, Intel Core processors expose one sensor per core; similarly, the ARM big.LITTLE SoC exposes one sensor per big core. These sensors are easily accessible on laptops or desktops through simple tools that export temperature information to userspace processes. On Android-based smartphones and tablets, apps can access the sensors without requiring any specific permissions. Compromised apps can therefore module system temperature and encode sensitive information in the temperature medium; posing a security risk.

2.2 Existing Temperature Integrity Algorithms

2.2.1 High Temperature Mitigation Strategies

The computing community along with the hardware vendors have attempted to mitigate the processor overheating problem using a combination of hardware and software approaches.
Hardware oriented solutions include capping of clock frequencies and integration of sophisticated power and temperature management features into the processor. These techniques to prevent processor overheating are already available, and some of these solutions are already found in the commercial state-of-the-art processors. One popular technique is to dynamically lower the processor's clock speed and/or voltages (DVFS) as the processor gets too hot. This has the effect of slowing down the processor which also lowers its temperature. However, this approach is usually reactive in nature resulting in an unforeseen and unplanned loss of performance. If the system operates under tight timing constraints, any performance degradation may have serious consequences on the overall reliability of the system. In addition, if the system uses a resource-constrained processor, the additional compute burden imposed by dynamic temperature management solutions may be unacceptable.
Software approaches attempt to control temperature by carefully controlling the workload being executed by the processor, either by shaping the incoming workload (e.g., by buffering some tasks), or by means of scheduling [**?**]. Among existing schemes that consider multi-core model, **?**] use integer linear programming to minimize the maximum temperature for a given set of real-time tasks. They use an equivalent circuit model to estimate the transient core temperatures. **?**] use the thermal characteristics of processing cores to derive preferred speeds for processing elements in a homogeneous multi-core system to minimize system temperature. **?**] focus on real-time task scheduling for processing elements without dynamic voltage scaling capability to target maximal power saving in a heterogeneous multi-core system.
Several approaches also attempt to address the peak temperature problem by bounding peak temperature for scheduling schemes geared towards meeting timing constraints. **?**] perform worst case temperature analysis for real-time tasks running on multi-core systems. They use the concepts of real-time calculus proposed by **?**], to model arrivals and computation demands of real-time tasks.

The worst case computation methodology is further used in **?**] to perform task and frequency assignment for real-time tasks executing on multi-core systems. The problem is formulated and solved as a binary optimization problem. **?**] perform worst case delay analysis on a stream of jobs following arrival and computation patterns based on real-time calculus. **?**] consider scheduling of periodic tasks on a multi-core system under soft thermal constraints. They design a proactive peak temperature manager which periodically estimates peak processor temperature. It uses machine learning to predict high temperature on a given core. If a core is predicted to overheat, dynamic power management techniques are applied to cool down the core without violating task timing constraints. A more recent software oriented approach relies on selectively using cores in the given multi- or many-core processor which have special spatial characteristics, e.g., ensuring that no heat dissipating core (i.e., a hot core) has a hot neighbor, in what is known as Dark Silicon Patterning [**?**]. Yet another method is to migrate tasks between a set of similar processing elements in order to avoid overheating any given element [**?**].

2.2.2 Temperature Related Security Risks

Integrity of data is a well researched topic which has often been addressed in the past. **?**] already analyzed this in 1973, by defining the confinement problem and describing how to exploit restricted data using covert channels. Here, a differentiation between covert channels and side channels has to be made. While the covert channel is used to actively transfer data over a channel, which is not visible to the user - hence covert; in side channels an indirect effect is used to gain knowledge about some restricted data. Now, with the breakdown of the Dennard scaling, analyzed by **?**], and the introduction of new thermal management technologies in Multi-Processor System-on-Chip (SoC) (MPSoC), thermal data can be used to build a covert channel or launch a side channel attack and threaten the integrity, availability, and confidentiality of data.
?] showed that it is possible to extract the private key of an RSA implementation on an AVR microcontroller. To do so, they operated the chip outside its specified temperature operation range and measured the temperature of the chip directly on the surface of the silicon. This way the Hamming weight of the processed data was leaked by the chip and could be used to reconstruct the used private key after several runs. Other work presented the possibility of side channel attacks on systems. **?**] demonstrated that it is possible to launch Denial-Of-Service (DOS) attacks on systems by creating hotspots on the silicon. This triggers the DTM which leads to severe loss of performance and results in the DOS.
In different studies, the implementation of covert channels by using temperature effects has been proven. Exploiting the local clock skew introduced by temperature variations is a well studied version of a covert channel based on temperature effects has already been shown by **?** **?**] and **?**]. For example this technique can be used to exploit services running in a hidden network, for example Tor. To do so, the attacker induces a load pattern to the server through sending a lot of requests to the server which causes temperature variations. By simultaneously observing the clock skew of all the candidate servers, the attacker can then reveal which server is hosting this particular service. **?**] also evaluated the capacity of this covert channel at approximately 20.5 bits per hour. Another side effect of temperature variations that can be used to compromise data integrity has been studied by **?**]. They encoded data in the fan speed and which forced the system to change the fan speed according to temperature changes that was inferred into the device by load management.
Not only is it possible to use side effects of temperature variations to exploit data, it has also been shown that active data transmission is possible using temperature. **?**] implemented a system which is capable to transfer data between two air gapped desktop systems only by observing the temperature. For this only the standard temperature sensors of the desktop systems were used. One desktop computer launched a load application to generate temperature variations, while the other one concurrently observes possible temperature variations. With this scheme they were able to send simple commands from one desktop computer to another one. Other work also explored if it is possible to send data over temperature within the package. One variant of such a covert channel is studied in

a Field Programmable Gate Array (FPGA) by **?** **?**] and **?**]. The FPGA was configured with two isolated components, whereas one part of the logic was configured as the thermal sender and the other one as the thermal receiver. **?**] on the other hand, studied the possibility to transfer data within a MPSoC from one core to another one. In this basic study they achieved up to 1.33 bits per second at an error rate of $11\,\%$ for the neighboring core. Furthermore they showed that it is possible to exchange information over the temperature channel between two applications which are running on the same core but not at the same time.

2.3 First Results and Research Directions

In SAFURE we will cater to both timing and security threats posed by temperature.

2.3.1 Temperature Analysis of Mixed-Criticality Systems

Temperature constraints in the domain of mixed-criticality scheduling have not been studied. This correlates to executing tasks of different safety requirements on a shared hardware platform. Requirement for this study is stated in Description of Action (DoA) pg 15. This research thread jointly considers temperature integrity and timing integrity of a mixed-critical system.

2.3.1.1 Task Model and Scheduling

In this project, we will adopt a standard dual-criticality task model [**?**]. Given is a set of n sporadic tasks scheduled on a uniprocessor: $\tau = \{\tau_1, \cdots, \tau_i, \cdots, \tau_n\}$. Each task may issue an infinite number of jobs. Any task τ_i is characterized by the minimum inter-arrival time between consecutive jobs W_i, relative deadline D_i, and criticality level χ_i. We assume constrained deadlines for all tasks: $\forall \tau_i \in \tau, D_i \leq W_i$. A task can either have high (HI) criticality or low (LO) criticality: $\forall \tau_i \in \tau, \chi_i \in \{\text{HI}, \text{LO}\}$. We denote all χ criticality tasks as τ_χ, i.e. $\tau_\chi = \{\tau_i | \chi_i = \chi\}$.
The Worst Case Execution Time (WCET) of tasks is modeled on both criticality levels with the WCET on HI criticality being more pessimistic than those on LO criticality [**?** **?** **?** **?**]. This model of criticality dependent WCET that is assumed by several papers in the research community has been challenged [**?**] [**?**] as non corresponding to the needs of the current industrial practice. In our opinion, this model could still be relevant/useful considering that it has been becoming increasingly difficult to characterize/bound WCET, especially in modern multi-core platforms (Refer to Section 4.1.3 for explanation). In this context, different WCET correspond to different levels of safety margins; where schedulability of HI criticality tasks is guaranteed with high safety margin.
Different levels of timing assurance are modeled in the system abstraction. We denote the WCETs of τ_i on criticality χ as $C_i(\chi)$. In addition, we assume that LO criticality tasks are not allowed to exceed their LO criticality WCETs. Based on the above model, we now introduce the two important mixed-criticality mechanisms regarding service adaptation and timing safety preparation.
Service adaptation. To improve system resource efficiency, conventional mixed-criticality scheduling techniques [**?**] provide dynamic guarantees to all tasks, which can be specified by a simple *scheduling mode* switch protocol:

- The system starts with LO scheduling mode, where all tasks are guaranteed to meet their deadlines providing that they do not exceed their LO criticality WCETs.
- If any HI criticality task exceeds its LO criticality WCET, the system transits immediately to HI scheduling mode, and the services provided to LO criticality tasks are *degraded* hereafter (or they are *terminated* in the extreme case) to protect the timeliness of HI criticality tasks.

For each LO criticality task τ_i, the degraded service is interpreted as the increased inter-arrival time W_i and/or deadline D_i. For notational convenience, we denote the parameters of task τ_i in mode χ as: $\{W_i(\chi), D_i(\chi), C_i(\chi)\}$.

Timing safety preparation. Intuitively, for any HI criticality task τ_i, if jobs of this task finish too "late" in LO scheduling mode, then there would not be enough time left until their deadlines to host extra workload when the system switches to HI scheduling mode. In light of this, *timing safety preparations* are proposed in the literature [**?** **?**]. The essential idea is to let HI criticality tasks finish earlier in LO scheduling mode so that enough CPU time is left to handle task overrun. For fixed-priority scheduled tasks, this can be ensured by guaranteeing shortened LO scheduling mode worst-case response times (WCRTs) for HI criticality tasks [**?**] (i.e. by increasing their assigned priorities). For Earliest Deadline First (EDF) scheduled tasks, the equivalence is to artificially shorten deadlines of HI criticality tasks in LO scheduling mode so that they are forced to finish earlier [**?**]. In both cases, we use $D_i(\mathsf{LO})$ to uniformly denote the timing safety preparation, i.e. $D_i(\mathsf{LO})$ denotes the WCRT of τ_i or its shortened deadline, both in LO scheduling mode.
Based on our model and assumptions, we have:

$$\begin{aligned} &\forall \tau_i \in \tau_{\mathsf{HI}} : W_i(\mathsf{HI}) = W_i(\mathsf{LO}) = W_i, \\ &D_i(\mathsf{LO}) < D_i(\mathsf{HI}) = D_i, C_i(\mathsf{HI}) \geq C_i(\mathsf{LO}), \end{aligned} \tag{2.1}$$

$$\begin{aligned} &\forall \tau_i \in \tau_{\mathsf{LO}} : W_i(\mathsf{HI}) \geq W_i(\mathsf{LO}) = W_i, \\ &D_i(\mathsf{HI}) \geq D_i(\mathsf{LO}) = D_i, C_i(\mathsf{HI}) = C_i(\mathsf{LO}). \end{aligned} \tag{2.2}$$

Notice that if LO criticality tasks are terminated in HI scheduling mode, then:

$$W_i(\mathsf{HI}) = +\infty, D_i(\mathsf{HI}) = +\infty, \ \forall \tau_i \in \tau_{\mathsf{LO}}. \tag{2.3}$$

2.3.1.2 Maximum Temperature Analysis

We assume a processor with two *processor states* (active and idle). Whenever the processor is processing jobs, it is in the active state. Otherwise it is automatically switched to the idle state. Due to increased leakage power consumption in the sub-nm era, we adopt a common linear approximation of the processor power consumption P as $\phi T + \psi$, where the leakage power depends on the system temperature T and ϕ and ψ are constants. Since the processor power consumptions are different for different processor states, we distinguish between power parameters in active state (ϕ_{act}, ψ_{act}) and those in idle state (ϕ_{idl}, ψ_{idl}).
We approximate the heat flow in the system by Fourier's Law. This yields a standard thermal model expressed in a first order differential equation:

$$\Omega \frac{dT}{dt} = -G(T - T_{amb}) + P, \tag{2.4}$$

where Ω, P, G and T_{amb} denote thermal capacity, power consumption, thermal conductance and ambient temperature, respectively. This leads to the closed-form solution to the above differential equation [**?**]:

$$T(t) = T^{\infty} + (T(t_0) - T^{\infty}) \cdot e^{-a \cdot (t - t_0)}, \tag{2.5}$$

where $T^{\infty} = \frac{GT_{amb} + \psi}{G - \phi}$, $a = \frac{G - \phi}{\Omega}$ and the system is in one processor state in $[t_0, t)$. Due to the different power consumptions in active and idle states, the complete system thermal model is characterized by two pairs $(T^{\infty}_{act}, a_{act})$ and $(T^{\infty}_{idl}, a_{idl})$.
Based on the above thermal model, we base our analysis on the techniques proposed in [**?**], where a constructive approach is proposed to identify a temperature-critical execution trace given bounded jobs arrivals and demands. In particular, it has been shown in [**?**] that, given the arrived demand bound function (adb$(\cdot)$) of all tasks on a fully available processor, the worst-case processed demands by the processor is bounded by:

$$\gamma(\Delta) = \inf_{0 \leq \delta \leq \Delta} \{\mathsf{adb}(\delta) + \Delta - \delta\}. \tag{2.6}$$

Furthermore, for any time instant t^*, the temperature-critical concrete execution trace of processed demands ($\gamma^*(0,\Delta)$, denoting the received executions in $[0,\Delta)$), which leads to the highest temperature at t^*, is linked to $\gamma(\cdot)$ as follows:

$$\gamma^*(0,\Delta) = \gamma(t^*) - \gamma(t^* - \Delta),\ \forall 0 \leq \Delta \leq t^*. \tag{2.7}$$

[**?**] provides further results on bounding the maximum checked time instant t^* for finding the peak system temperature. We refer the interested readers to [**?**] for more details. To summarize, we base our analysis on the following work flow:

$$\left.\begin{matrix} \text{adb}(\Delta) \xrightarrow{(2.6)} \gamma(\Delta) \xrightarrow{(2.7)} \gamma^*(0,\Delta) \\ T_0 \end{matrix}\right\} \xrightarrow{(2.5)} T_{peak}. \tag{2.8}$$

Notice that for the last step, we need the initial temperature of the system T_0, and the temperature-critical execution trace is simulated according to (2.5) to derive the worst-case temperature.
In SAFURE we will perform the analysis of the peak system temperature under common mixed-criticality scheduling techniques based on Fixed Priority (FP) or EDF. Since the system may switch scheduling mode during runtime, it is natural to perform peak temperature analyses separately for different scheduling modes.

2.3.2 Temperature Threats to Platform Security

In this research thread, we study the use of temperature as a covert communication channel. This correlates to executing tasks of different security requirements on a shared hardware platform. Requirement for this study is stated in DoA pg 15.

2.3.2.1 The Covert Channel Threat Model

Covert channels can broadly be classified as *storage* or *timing* channels. We study storage channels, where the source app directly or indirectly writes to a shared resource, which the sink app reads [**? ?**].
We are interested in the scenario introduced in the example of Figure 2.2. Without loss of generality, we assume that the sink app just records a temperature trace by reading the sensors and later sends it to the attacker over the network; message decoding is done offline. Thus, the sink app is mostly idle and only periodically wakes up to read the sensor.

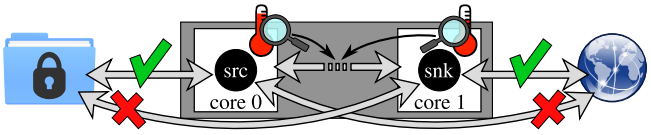


Figure 2.2: The source app (src) has access to restricted data but no network access; the sink app (snk) has no access to the restricted data but has network access. A compromised source app can leak sensitive data to the sink app through the thermal covert channel, breaking privilege separation.

We target devices where the Operating System (OS) puts idle cores to sleep and, when sleeping, cores consume close to zero power and produce almost zero heat. We note that the mobile devices that we target are idle or lightly-loaded most of the time (e.g., a smartphone resting in a pocket or a laptop just running a text editor). Thus, the source and sink app can check when the system load is low to start using the covert channel, so as to avoid interference.
Finally, we note that modern mobile multi-cores, e.g., Intel Core mobile processors or ARM multi-cores, generally feature one temperature sensor per core and that these sensors are easily accessible by userspace processes or apps. For instance, on Linux, `lm_sensors` exports a simple command-

line interface; on Windows, CoreTemp offers a graphical interface. While setting up these tools might require administrative rights (e.g., `# sensors_detect`), they are commonly installed on client devices. On Android devices, the temperature sensors are even easier to access for the apps: we verified that the CPU-Z app (v. 1.15), available on the Google Play Store, requires no system permissions to be installed and it reports several temperature measurements on a Nexus 4 running Android 5.0.2. Moreover, once the sensors are exposed, any app can normally read all sensors, regardless of which core it is running on.

2.3.2.1.1 Analysis Methodology

Following Shannon's seminal work [**?**], researchers extensively studied ways to determine the capacity of a wide range of channel models [**?**]. Still, even with this vast theoretical literature available, estimating the capacity of a physical channel remains very challenging: it requires using an appropriate model and retrieving quantitatively accurate measurements of the channel parameters, despite of noise and limited precision. We tackle this challenge by leveraging the simple model and determining its transfer function $H(f)$ through carefully designed experiments based on the experimental setup. We can search, among all the possible input patterns $x(k)$, the one that has the frequency characteristics that make the most information pass through the channel; in other words, we need to find the best allocation of the input power $\hat{S}_{xx}(f)$ across the frequency spectrum. The key aspect in this method is that we can only allocate as much power as we are able to put into our input signal, i.e., we have a power cap p_0. The general approach to determining $\hat{S}_{xx}(f)$, and thus C (channel capacity), subject to a power cap p_0 is known as *water-filling* [**?** **?**] *Water-filling* is based on the assumption that the optimal input spectrum is the one that allocates power such that the sum of the noise and the signal power is constant over the whole channel spectrum; so more power of the signal is in parts of the spectrum with high Signal to Noise Ratio (SNR). We study two different solutions based on this technique. First, we consider the *classic* solution [**?**], which considers the constraint p_0 on the average input power. Second, we analyze a *constrained-input* solution [**?**] that explicitly considers the additional constraint that the input to our channels is a binary value (active/idle).

Classic water-filling approach The classic water-filling technique allows to compute the capacity of channels with arbitrary transfer function $H(f)$ and additive gaussian noise $q(k)$, not necessarily white [**?** **?**]. White noise implies that the noise has a constant power spectrum $S_{qq} = N_0$ across the frequency range A_λ. If we can estimate the power spectrum of the channel $S_{hh} = |H(f)|^2$ and of the noise S_{qq} then, given a cap p_0 on the average input power, we can derive the channel capacity according to Equation 2.9 [**?** , Eq. (6.15)].

$$C_b = \max_{S_{xx}} \left\{ \int_{\mathcal{F}} \log_2 \left(1 + \frac{S_{xx}(f) \cdot S_{hh}(f)}{S_{qq}(f)} \right) df \right\} \text{ [bps]}, \tag{2.9}$$

$$\text{under the constraint that } \int_{\mathcal{F}} S_{xx}(f) df \leq p_0 \tag{2.10}$$

The capacity C_b is determined by the *spectral power allocation* $S_{xx}(f)$, which cannot exceed the power cap p_0, as Equation 2.10 states. We can maximize the expression in Equation 2.9 and determine the capacity by intelligently shaping the power allocation S_{xx} so that more power is allocated at those frequencies with better SNR. This ideal allocation $\hat{S}_{xx}$ can be determined with a *water-filling* procedure [**?** **?**], which we do not describe in details here. As it is shown in **?**], we are able to estimate S_{hh} and S_{qq} for our channels; thus, we can use the water-filling procedure on Equation 2.9 to estimate the capacity C_b. We expect C_b to be an upper bound on the real capacity C, because the classic water-filling approach does not consider the more stringent constraint that our input is required to be a binary value.

Constrained-input water-filling In a 1992 paper, **?**] studied the capacity of *saturation recording*, i.e., the capacity of storage systems such as tape recorders or optical disks. While this problem has, in general, little to do with our study, it has the same *saturation* constraint on the channel input: input values can only be either 0 or 1. This shared property allows us to leverage their expression for an

upper bound C_a on the channel capacity C [**?** , Eq. (11)]. We report this result (with minor notation changes) in Equation 2.11.

$$C \leq C_a = \max_{\lambda} \left\{ \frac{1}{2} \int_{A_\lambda} \log_2 \left(\lambda \cdot S_{hh}(f) \right) df \right\} \tag{2.11}$$

$$\frac{1}{2} \int_{A_\lambda} \left(\lambda - \frac{1}{S_{hh}(f)} \right) df \leq \frac{p_0}{N_0} \tag{2.12}$$

The parameter λ must be maximized subject to the constraint of Equation 2.12, which makes sure that the SNR does not exceed the ratio of the power cap p_0 over the noise power N_0. These equations assume that the noise is white. Since, in our channels, S_{qq} is not constant, we use this constrained-input solution only after splitting the channel into sub-bands where S_{qq} can be assumed constant; more details on this technique can be found in **?**]. Finding the λ that maximizes Equation 2.11 subject to Equation 2.12 follows again a water-filling procedure.

2.3.2.1.2 Experimental Setup

We base our analysis on experimental data collected from two diverse and representative hardware platforms:

1. a Lenovo ThinkPad T440p laptop, featuring a quad-core Intel Core i7-4710MQ processor clocked at 2.5 GHz;
2. an Odroid-XU3 board, featuring a Samsung Exynos 5422 SoC including an ARM big.LITTLE processor with two quad-core clusters of Cortex-A7 and Cortex-A15 cores, respectively. The big cluster is clocked at 2.1 GHz.

In the rest of this sections, we refer to platform 1 as *Laptop* and to platform 2 as *Smartphone*. *Laptop* is representative of current business laptops; *Smartphone* is representative of hand-held devices (it has the same SoC as the Samsung Galaxy S5 SM-900H smartphone).

System settings. On both *Laptop* and *Smartphone*, we install Ubuntu 14.04.2 and we use the `/dev/cpu_dma_latency` interface of the Linux kernel to limit the maximum wakeup latency to $10\,\mu$s. On *Laptop*, the temperature sensors are refreshed every 1 ms [**?**]. We were not able to find the sensors refresh period for *Smartphone* on the SoC documentation. To determine this parameter, we collected several traces with varying system load, using 1 ms as the sampling period; we noticed that the temperature never changed more often than every 5 ms, which we take as the sensor refresh rate for this platform. Based on these characteristics, we set the sampling period to $T = 1$ ms for *Laptop* and $T = 5$ ms for *Smartphone*. Therefore, the Nyquist frequency of our discrete system is $0.5/1$ ms$= 500$ Hz for *Laptop* and 100 Hz for *Smartphone*.

To favor repeatability, we run all experiments in a controlled, while still realistic, environment. We set both devices in an air-conditioned server room with an ambient temperature of ≈ 23 C$^\circ$ and, for both, we fix the fan speed to the maximum level and set the clock frequency of active cores to the maximum. In order to avoid scheduling artifacts, we run the source and sink app with the `SCHED_FIFO` scheduling class at highest priority by using the `pthread_setschedparam()` interface and we pin the source app to one core by using the `pthread_setaffinity_np()` interface. During all experiments, the system is idle except for the source and sink apps and the default system services of the Ubuntu installation. We run the source app on the third core in the array, i.e., on core 4 on *Laptop*, which has eight virtual cores with two-way hyper-threading, and on core 6 on *Smartphone*, where cores 0 to 3 are the LITTLE cores and cores 4 to 7 are the big cores. In the rest of this section, we only count the four physical (big) cores, starting from 0; thus, for both platforms, we say that we run the source app on core 2 and we record the temperature traces from cores 0 to 3. On *Laptop* we exploit hyper-threading and run the sink app on the odd-numbered virtual cores and on *Smartphone* on the first LITTLE core to avoid timing interference. Unless differently specified, we use these settings in all our experiments.

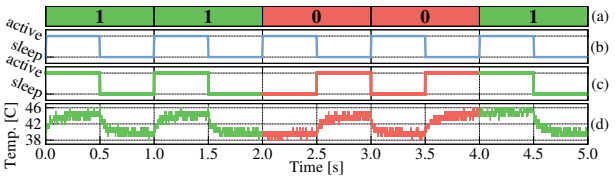


Figure 2.3: An input message (a), encoded onto the 1 Hz clock (b), gives the execution trace (c), which leads to the temperature trace (d) on the same-core channel of *Laptop*.

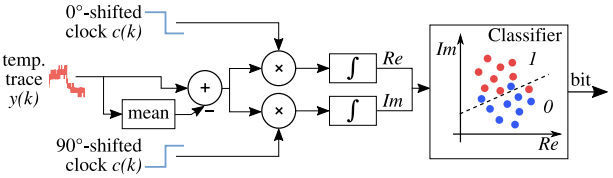


Figure 2.4: Block diagram of our bit-wise decoding scheme.

Encoding and decoding scheme. A simple way to keep the channel in the dynamic range during communication is to encode the input message so as to maintain, on average, a constant load. To do so, we use square waves with a 50% duty cycle as a *clock* signal onto which we *encode* the input message. We use the *Manchester encoding* scheme [**?**] to generate the execution trace of the source app, as Figure 2.3 illustrates for a 5-bit message and a 1 Hz clock.
A *one* in the message is encoded into an unmodified clock signal in the execution trace; a *zero* becomes a $180°$ phase-shifted clock signal in the execution trace. The resulting execution trace leads to temperature traces oscillating around a roughly constant average, as Figure 2.3 (d) shows for the same-core channel on *Laptop*. The transmission rate directly depends on the frequency of the clock signal, since the trace carries 1 bit of information per period of the clock, i.e., r bps for a r Hz clock.
Message decoding happens offline from the temperature traces recorded by the sink app. The first step of decoding is determining the phase of the clock signal. For simplicity, we synchronize our experiments so that the beginning of the temperature trace coincides with the beginning of the message. In a real attack, where this synchronization would not be possible, the source app could send a known preamble that can be used to detect the clock phase. Once the clock phase is detected, it will not change during an experiment, since our source and sink app are designed to not accumulate clock skew. To proceed with decoding, we look at each clock period, i.e., at each bit, separately. As Figure 2.4 shows, for each bit, we first get a 0-mean signal by subtracting its mean temperature; in this way, the decoding is robust against long-term temperature variations due to environmental changes. We decode the resulting trace with traditional signal-processing techniques [**?**]. We first multiply the trace with a $90°$ and a $0°$ phase-shifted clock signals and we integrate over the two resulting signals ($\int$ blocks in Figure 2.4). The two resulting numbers are the real (*Re*) and imaginary (*Im*) parts of a representation of the bit in the complex plane $\mathbb{C}$. To classify each bit as a 1 or a 0 in this signal space, we use a naïve-Bayes classifier [**?**] with a kernel smoothing density estimate[1], previously trained on data from the same platform.

2.3.2.1.3 Results

Theoretical capacity bounds. We can compute the two capacity bounds C_b and C_a, with the classic and constrained-input water-filling methods, respectively. Since we work with discrete spectra, we accordingly adapt the equations of paragraph 2.3.2.1.1 to use summations instead of integrals and to consider the discretization intervals along the frequency range. Figure 2.5 shows the capacity bounds C_b (left) and C_a (right) that we compute with the classic and constrained-input water-filling methods, respectively. As expected, $C_b > C_a$ and the bound for the same-core channel is the highest for both platforms and both methods. In general, the trend across the four channels seems consistent on the two platforms. These results do not exclude that the same-core channel might be a security threat.
Performance evaluation. To evaluate our transmission scheme, we encode several random messages onto clock signals at different frequencies and we use our source and sink app to transmit and record these messages on our two platforms, configured according to the reference setup described in paragraph 2.3.2.1.2. We decode the temperature trace from each channel with our classifier; as

[1] We use the `NaiveBayes` object of Matlab R2015a, with default settings.

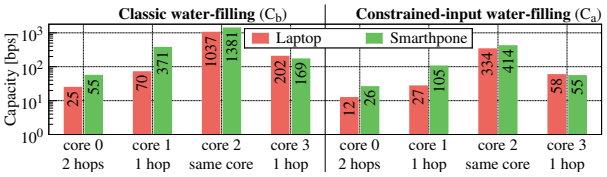


Figure 2.5: Upper bounds C_b (left) and C_a (right) on the channel capacity C for the four channels on *Laptop* and *Smartphone*. The y-axis is in logarithmic scale.

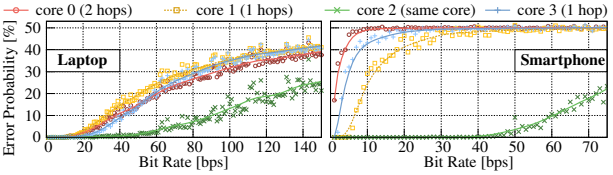


Figure 2.6: Error probability on decoding a 5000 bit random message for the four channels on *Laptop* and *Smartphone*, for transmission rates up to 150 bps and 80 bps, respectively.

the performance indicator, we use the error probability, as measured through the empirical bit error rate, i.e., the relative number of misclassified bits. We just report raw transmission rates and error probabilities and do not evaluate error correction strategies; we leave such study to future work.
Figure 2.6 shows the resulting error probability (measurements and bezier trends) for the four channels on our two platforms. For both *Laptop* (left) and *Smartphone* (right), the same-core channel shows very few errors ($\ll 1$%) up to ≈ 40 bps. Up to this rate, *Smartphone* performs better than *Laptop*, thanks to the much lower noise. Instead, at increased rates, errors increase more slowly on *Laptop*, where we achieve ≈ 90 bps at 10% error probability, than on *Smartphone*, where the rate is ≈ 60 bps at the same error level. *Laptop* shows better performance also for the 1-hop and 2-hop channels, where the error probability remains very close to 0 up to ≈ 10 bps and hits the 10% level between 30 bps and 40 bps. On *Laptop*, the 2-hop channel does not perform much worse than the 1-hop channels; instead, on *Smartphone* the error probability immediately increases steeply and the performance gaps are more evident, with the 2-hop channel showing several errors already at 1 bps. These results relate with the stronger quantization effect and the higher attenuation for these two channels on *Smartphone*.
Sensitivity to environmental conditions. Finally, we evaluate how variations in the environmental conditions affect the error probability on our channels. We identify four important parameters that, in a real attack, would not be fixed as in our initial experimental setup and we evaluate the sensitivity of our results to variations on these parameters. As a representative case, we show the results of this study on the same-core channel on *Laptop*. Figure 2.7 shows how the error probability is affected when changing these four parameters in the experimental setup:

1. Setting the fan speed to automatic (Fan auto);
2. not pinning the apps to a specific core (No pinning);
3. using the default Linux scheduling policy instead of the high-priority `SCHED_FIFO` (No RT);
4. using the conservative DVFS governor which changes the cores frequencies (DVFS Conserv.).

These four parameters have different impact on our baseline results, represented in Figure 2.7 by the solid red line. An automatic fan speed highly affects the channel and makes it very chaotic. This is intuitive as the fan speed control is designed to keep the temperature on a low constant level, which contradicts the intention to code data in temperature variations. Like an automatic fan speed, enabling DVFS has a high effect on the communication channel and therefore a communication on the same-core channel highly unstable. This is because the active frequency of the cores largely determines the active power consumption, and thus temperature.
Dropping real-time priority significantly affects the error probability only for rates faster than ≈ 15 bps. When the source and sink apps are not pinned to a specific core, the different channels effectively move with the source app. As an example, Figure 2.8 shows part of a trace from *Smartphone* where the source app, which is transmitting a 1 Hz clock signal, migrates between cores 1 and 2. Initially, reading the temperature from core 2 corresponds to a same-core channel, while it becomes a 1-hop channel at time ≈ 1.5 s, when the source app migrates to core 1. As Figure 2.9 shows, if the sink app always observes the same core (core 2 on *Laptop* in this case), the error probability without thread

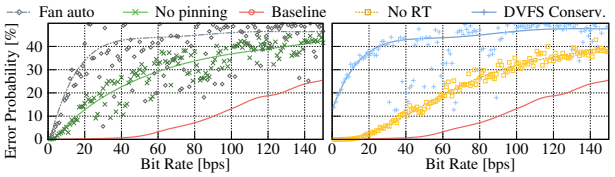


Figure 2.7: Sensitivity of the error probability to using automatic fan speed, not pinning the apps to cores, no real-time scheduling, or the conservative Linux DVFS governor.

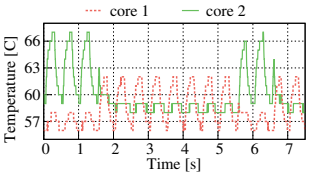


Figure 2.8: Traces from cores 1 and 2 of *Smartphone*; the source app is not pinned.

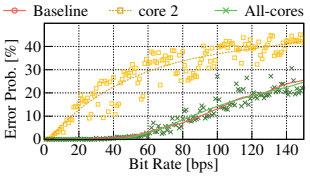


Figure 2.9: Same-core vs. all-cores channel comparison with no pinning on *Laptop*.

pinning will sensibly increase compared to the baseline, since the channel type keeps changing. However, there is a simple way to work around this issue. Since the sink app can always read the temperature of all the cores, we can simply look at the *all-cores channel*, which is the sum of the temperatures from all cores. As Figure 2.9 shows, the all-cores channel has performance comparable (or possibly better) than the same-core channel.

We conclude that our communication scheme is robust to disabling thread pinning and, to some extent, to dropping real-time priorities and having background system load. The most sensitive parameters are varying fan speed and enabling the DVFS governor, which makes communication impossible with our scheme but might enable a different covert channel when all cores share the same active frequency.

Chapter 3 Data Integrity

The following chapter focuses on data integrity measures. It provides an overview of current state-of-the-art techniques to preserve data integrity and additionally introduces first results of data integrity algorithms for safe and secure systems in the context of SAFURE.
The amount of data in today's automobiles is enormous and safe and reliable operation of a car strongly depends on the integrity of these data. In a classical vehicle, actuators rely on sensor data and exchange commands via the internal bus system of the car. In nowadays vehicles, additional infotainment data, such as maps, driver's personal data, and other media data, is stored and processed in the vehicle. Therefore, strong data integrity measures are a stringent requirement in automotive use cases. The goal of integrity algorithms is to protect data from unauthorized modifications also comprising data generation and deletion. Data integrity can be divided into the following major aspects [**?**]:

- unauthorized data modification
- unauthorized data insertion
- unauthorized data replay
- unauthorized data generation
- unauthorized data deletion

Data integrity algorithms clearly enhance security as they allow the detection of unauthorized and unwanted change to data. Therefore, any critical system where data integrity plays a major role requires integrity measures.
With electronic systems taking on various safety relevant tasks, enhanced security protection has a clearly positive impact on safety. However, integrity mechanisms add an additional processing delay which needs to be considered in the implementation of a time-critical system, e.g. in the automotive, avionics or healthcare domain.
There are various measures to verify and protect data integrity, the most commonly used ones are presented in the following sections. In Section 3.1 an overview on non-cryptographic checksums is given. To detect the intentional manipulation of data, cryptographic integrity protection techniques are required. These techniques can be separated in symmetric approaches, i.e. MACs, presented in Section 3.2.1 and asymmetric measures, i.e. digital signatures, discussed in Section 3.2.2. For both types of cryptographic measures, different example algorithms are presented. Finally, a short introduction to Public Key Infrastructures (PKIs) is given in Section 3.3 as this infrastructure is essential for an efficient and secure usage of asymmetric cryptography in field.

3.1 Checksums and Error Correcting Codes

Checksums are used by various systems to provide protection against unintentional or accidental errors occurring during transmission over communication channels. The receiver calculates the checksum over the received data and verifies this value against the received checksum. There are different types of checksums varying from simple additions to more complex Cyclic Redundancy Check (CRC)

algorithms, which can detect more errors but are also computationally more expensive. Furthermore, error correcting codes cannot only detect errors but are also able to correct some errors by introducing more redundancy to the transmitted message.
Checksums and error correcting codes are required to fulfill safety requirements such as those listed in Part 6 of ISO 26262 [**?**]. However, there is no secret key necessary to calculate a checksum, i.e. an adversary may not only alter the message but also the corresponding checksum to masquerade the manipulation. Therefore, a cryptographic approach as described in the subsequent sections is needed in order to detect targeted attacks.

3.2 Authentication

Integrity protection may be based on symmetric authentication techniques. In that case, the same secret key is required for generating as well as verifying a Message Authentication Code (MAC), which is a cryptographic checksum that authenticates a message. In general, hash functions or symmetric encryption algorithms are used to generate MAC values. However, as the knowledge of a common secret key is needed in order to generate and verify the value of the MAC, all network validating components must be equipped with the same key. This can lead to the key distribution problem: the number of keys required to be distributed through a secure channel grows exponentially with the number of network components, because each pair of components needs to agree on an individual key (resulting in a number of keys that is quadratic to the number of components).
Alternatively, asymmetric cryptography (i.e., digital signatures) can be used to protect the integrity and authenticity of data, avoiding the key distribution problem, because in this setting only the public key of each component needs to be distributed to all components (resulting in a linear number of keys). However, asymmetric cryptography algorithms are computationally costly (i.e., about 1000 times slower than MACs). Furthermore, the key sizes for some asymmetric algorithms (e.g. RSA – with the exception of Elliptic Curve Cryptography (ECC) algorithms like ECDSA) are significantly larger than their symmetric counterparts. Table 3.1 gives an overview of recent recommendations for minimum key lengths for different data integrity algorithms that are going to be at the center of this chapter and work package. It is interesting to observe that a MAC based on the symmetric encryption algorithm AES requires a 128bit key to achieve the same cryptographic strength as a RSA signature with a key length of 3072 bit according to [**?**].

Institution	Year	RSA	DSA	ECDSA
ECRYPT II [**?**]	2012	2048 bit	q>=256 bit, p>=3072 bit	256 bit
NSA [**?**]	2015	3072 bit	-	384 bit
NIST [**?**]	2016	3072 bit	q>=256 bit, p>=3072 bit	256-383 bit
BSI [**?**]	2016	2000 bit	q>=250 bit, p>=2000 bit	250 bit

Table 3.1: Key Length Recommendations

3.2.1 Message Authentication Codes

MACs are based on symmetric algorithms that use the same key for generating as well as verifying the integrity value. A MAC value is calculated either using a hash function (Keyed-Hash MAC (HMAC)) or a symmetric cipher (CMAC). As only the knowledge of the secret key permits the generation and verification of the MAC, this key must be shared with all possible validators.

The basic principle of MACs is illustrated in Figure 3.1: Bob generates the MAC of the message $\mathtt{x}$ using a the shared symmetric key $\mathtt{k}$. He sends the message $\mathtt{x}$ and the MAC y to Alice who can calculate the MAC $\mathtt{y'}$ of the message she received. If $\mathtt{y'} = \mathtt{y}$, she knows that she has received exactly the message Bob sent.

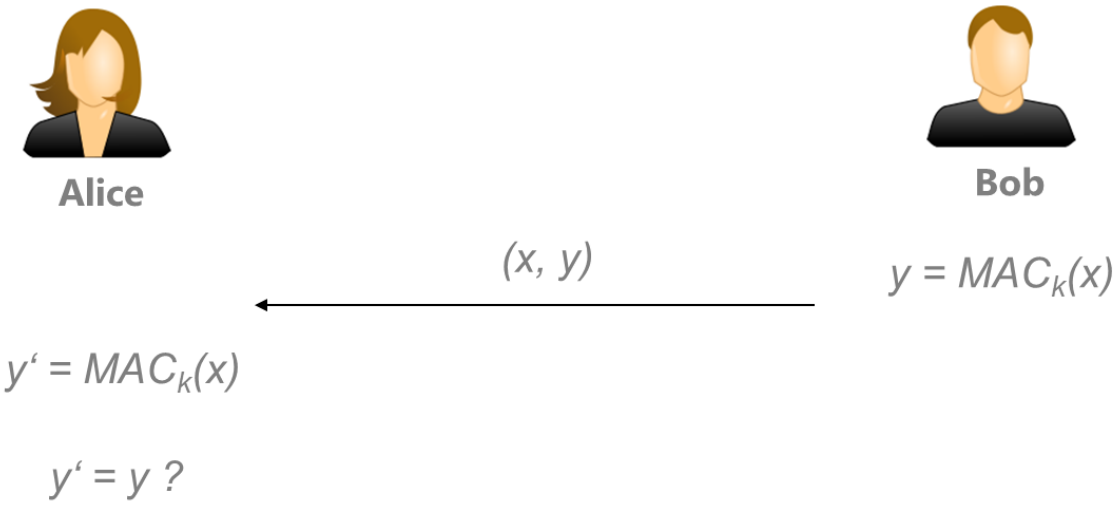


Figure 3.1: Basic principle of Message Authentication Codes (MACs)

Hash-based message authentication codes (HMACs)

HMACs use a cryptographic hash function and a secret key to generate the checksum. Generally, the setup of an HMAC looks as follows [**?**]:

$$\texttt{HMAC}(\texttt{K}, \texttt{m}) = \texttt{H}(\texttt{K} \oplus opad \mid\mid H(\texttt{K} \oplus ipad \mid\mid \texttt{m})),$$

where:

- `H` is a cryptographic hash function (e.g. SHA-256)
- `K` is a secret key
- `m` is the message
- $opad$ is the outer padding (e.g., the byte `0x5C` repeated until block length is reached)
- $ipad$ is the inner padding (e.g., the byte `0x36` repeated until block length is reached)
- $\oplus$ indicates an exclusive disjunction (XOR) operation
- $||$ indicates the concatenation of two bit strings

The strength of the procedure depends on the strength of the underlying hash function. In [**?**] and [**?**], the application of SHA- and MD5-based HMACs is further described.

Cipher-based message authentication codes (CMACs)

CMACs make use of a block cipher as well as a shared secret key. The most widely used CMAC are XCBC-based modes which are similar to the CBC-MAC, but with improved security features [**?**]. Cipher Block Chaining (CBC) refers here to the cipher block chaining mode as mode of operation of the block cipher.
The extended cipher block chaining modes offer enhanced security measures compared to CBC-MACs as well as several favorable features related to secrecy and integrity. These include the suitability for real-time message authentication and support of multiple encryption modes. In addition, they provide security against adaptive chosen-plaintext and message-integrity attacks, see [**?**].
In general, the setup of a CMAC looks as follows [**?**]:

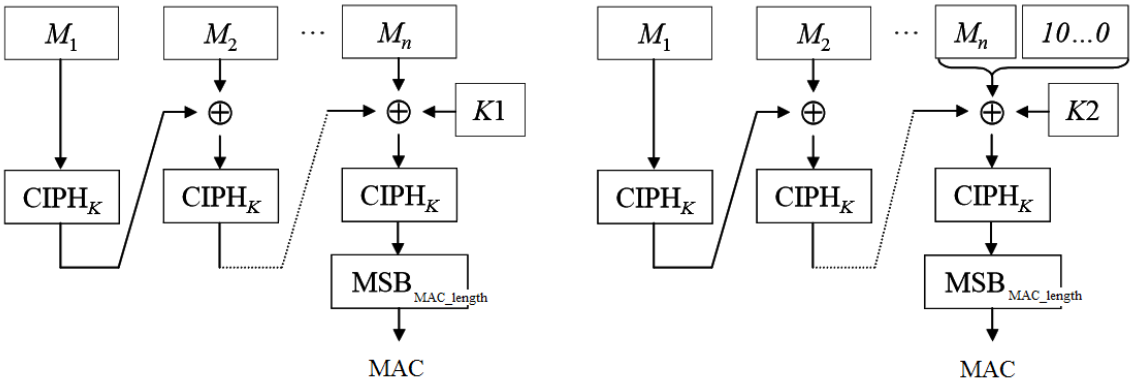


Figure 3.2: The two cases of CMAC Generation [**?**]

1. Generate the subkeys $\mathtt{K}_1$ and $\mathtt{K}_2$
2. Separate the message $\mathtt{m}$ in n blocks of length b
3. If the last block, $\mathtt{m}_n$, is a complete block, let $\mathtt{m}_n = \mathtt{K}_1 \oplus m_n$; else, let $\mathtt{m}_n = \mathtt{K}_2 \oplus (\mathtt{m}_n \mid\mid 10^j)$, where $j = nb - m_{\mathtt{length}} - 1$.
4. Let $\mathtt{c}_0 = 0^b$.
5. For $i = 1$ to n, let $\mathtt{c}_i = cipher_K(\mathtt{c}_{i-1} \oplus \mathtt{m}_i)$.
6. Return $\mathtt{MAC} = MSB_{\mathtt{MAC}_{\mathtt{length}}}(\mathtt{c}_n)$

where:

- $\mathtt{K}$ is a secret key
- $\mathtt{K}_1$ and $\mathtt{K}_2$ are secret subkeys derived from $\mathtt{K}$ as specified in [**?**]
- $\mathtt{m}$ is the message of bit length $m_{\mathtt{length}}$
- $cipher()$ is a secure block cipher (e.g. AES in CBC mode) with block length b (e.g. 128 for AES-128)
- $\mathtt{MAC}$ is the output of the algorithm of bit length $\mathtt{MAC}_{\mathtt{length}}$
- $\oplus$ indicates an XOR operation

This general algorithms is illustrated in Figure 3.2.

Poly1305

Poly1305 is a very fast CMAC algorithm based on a polynomial evaluation modulo $2^{130} - 5$. The first version is based on AES-128 [**?**], but it can also be used with other ciphers, such as Salsa20 [**?**] or ChaCha20 [**?**]. The major advantage of Poly1305 is that it provides a parallelizable high-speed algorithm with a low computational overhead per message, i.e. it is also efficient for short messages and not just optimized for long messages. The security of Poly1305 depends on the security of the underlying cipher. It has been standardized in RFC 7539 and is used – in combination with ChaCha20 – in the TLS/SSL protocol.

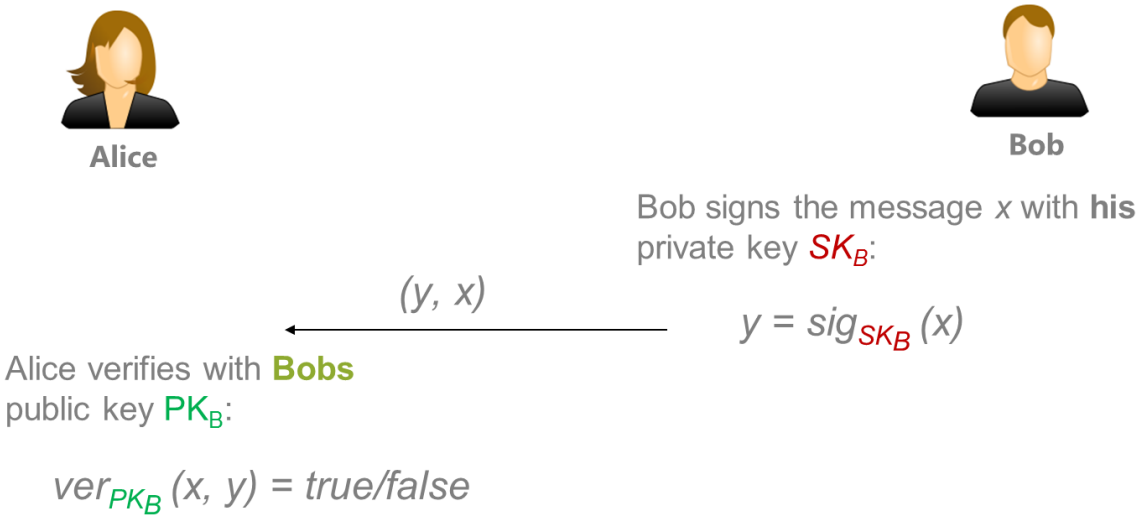


Figure 3.3: Basic principle of Digital Signatures

3.2.2 Digital Signatures

In addition to verifying data integrity and providing a means of authentication, digital signatures ensure non-repudiation of a signed message. Digital signatures mainly rely on a PKI which is commonly used for asymmetric cryptography. Details on advantages, challenges and requirements of PKIs are given in Section 3.3. In case of asymmetric cryptography, each entity, e.g. a device, person or email account, has its own key pair consisting of a public and a private key. The secret key is only known to the device itself whereas the public key is distributed to all other communicating parties.
The basic principle of digital signatures is illustrated in Figure 3.3: Bob calculates the signature y of the message x using his private key SK_{B}. He sends the message x and the signature y to Alice who does not know Bob's private key but his public key PK_{B}. Alice runs the verification algorithms on the message x and signature y she received and gets a Boolean output. If the verification algorithm returns `true` she knows that the message x has not been altered during transmission and that it has been signed by Bob - or someone who knows Bob's private key.
Digital signatures depend on the secrecy of the private key in order to prevent its forgery. In the following paragraphs, examples of digital signature algorithms are presented.

RSA signatures

RSA signatures are part of the RSA crypto system that has been proposed in 1978 [**?**]. Key generation and the operations for message signing and verifying are identical to the steps for message encryption and decryption.
For signature generation and verification, a key pair $PK_{\mathrm{B}}, SK_{\mathrm{B}}$ is required. To generate the key pair, two prime numbers p and q are chosen at random and multiplied $n = pq$. n is the modulus necessary for all signature and verification operations. Then, Euler's totient function, $\varphi(n) = (p-1)(q-1)$, is calculated. Formally, this function gives the number of totients of n, i.e. the number of integers i for which are coprime or relatively prime to n (i.e. $gcd(i, n) = 1$). Additionally, an integer e which needs to be coprime to $\varphi(n)$ and its modular multiplicative inverse $d = e^{-1} \mathrm{mod}\ \varphi(n)$ are chosen. The secret key $SK_{\mathrm{B}} = (n, d)$ is used for signature generation. The public key $PK_{\mathrm{B}} = (n, e)$ can be distributed and is required for signature verification. The parameters $p, q, \varphi(n)$ have to be kept secret, too, because d can be easily calculated if these values become public.
The signature s of a message m is computed as follows, where H is a hash function:

$$s := \mathrm{H}(m)^d \mathrm{mod} n$$

The receiver of (m, s) uses the corresponding public key to verify the signature. He checks if $\mathtt{H}(m) = s^e mod\ n$.
In practice, this plain RSA is not used because there are well-known chosen plain- and ciphertext attacks. Various padding techniques, such as the Probabilistic Signature Scheme (PSS), can be applied to the input of the RSA signature algorithm to provide additional security. Additionally, implementations of RSA need to consider side-channel and timing attacks. However, while the generation of signatures is relatively slow, the signatures can be verified fast and efficiently which is usually the more frequent operation in practice, for example a digital certificate is signed once but validated multiple times.

DSA

The Digital Signature Algorithm (DSA) has been proposed by the National Institute of Standards and Technology (NIST) [**?**]. Its security is based on the discrete logarithm problem, i.e. the problem that discrete logarithms cannot be efficiently computed in certain groups.
The signature algorithm uses a hash function $\mathtt{H}$, e.g. SHA-1 or SHA-2. Furthermore, two primes p and q are chosen in such a way that $p - 1$ is a multiple of q. The primes are N and L bit long, respectively. It is recommended [**?**] to choose $N >= 3072$ and $L >= 256$.
Additionally, a parameter g needs to be chosen such that the multiplicative order of $g \mathrm{mod} p$ is q. The parameters p, q, g are shared between all users of the system and each user derives his own key pair x, y as follows:

- The private key x is a random number with $0 < x < q$.
- The public key $y := g^x \mathrm{mod} p$ is calculated.

The for each new signature (r, s) of a message m a random value k where $0 < k < q$ is chosen. The signature is calculated as follows

$$r := (g^k \mathrm{mod} p) \mathrm{mod} q$$

$$s := k^{-1}(\mathtt{H}(m) + xr) \mathrm{mod} q$$

If either $r = 0$ or $s = 0$, a different random value k has to be chosen.
To verify the signature, $v = (g^{u_1} y^{u_2} \mathrm{mod} p) \mathrm{mod} q$ with $u_1 = \mathtt{H}(m) w \mathrm{mod} q$ and $u_2 = rw \mathrm{mod} q$ where $w = s^{-1} \mathrm{mod} q$ is calculated. The signature is valid if $v = r$. Signatures where $0 < r < q$ or $0 < s < q$ are not satisfied can be rejected.

ECDSA

The Elliptic Curve Digital Signature Algorithm (DSA) (ECDSA) [**?**] is an ECC variant of the DSA which has been described in the previous subsection. In contrast to the DSA, calculations are done on an elliptic curve. The security of the Elliptic Curve DSA (ECDSA) relies on the assumption that the Ellitpic Curve Discrete Logarithm Problem (ECDLP) is difficult. The choice of the elliptic curve determines the overall security of the system (since certain types of curves are known to be vulnerable to specific attacks) as well as the speed of the implementation. The major advantage of the ECDSA is that the key length necessary to achieve a comparable level of security is significantly shorter than for the DSA.

EdDSA

The Edwards-curve DSA (EdDSA) [**?**] is a variant of Schnorr signatures [**?**] based on Twisted Edwards curves, a special type of elliptic curves [**? ?**].
The fact that it is not necessary to create randomness for new Edwards-curve DSA (EdDSA) signatures but only for new session keys helps to prevent attacks comparable to the attack on the Sony PlayStation 3 security system [**?**].
The EdDSA can be implemented without conditional branches and array indexing based on secret

data in order to prevent side-channel attacks. Furthermore, it has been observed [**?**] that implementations of the EdDSA are significantly faster than recent implementations of most other signature schemes such as DSA or ECDSA. According to the publication, only certain RSA-based systems offer faster verification times but are slower at signing and require longer keys for a comparable level of security, cf. Table 3.1.
Due to these characteristics, the EdDSA is a promising signature algorithm for systems with high requirements regarding speed and security.

3.3 Public Key Infrastructure

PKIs are widely used in standard IT environments and in particular in the Internet but have also proven useful in embedded applications. In a classical PKI, the root Certificate Authority (CA) can issue and revoke certificates for sub-CAs. More precisely, the sub-CA generates a pair of private and public keys and sends the public key to the CA, which issues the certificate. The certificate embeds at least the following information: the identity of the issuing CA, the identity of the sub-CA, the public key of the sub-CA and a signature of said data generated with the private key of the CA. Consequently, the sub-CA is able to generate signatures on behalf of the CA: anyone who has the public key of the CA, the certificate of the sub-CA and a signature generated by the sub-CA is able to verify the signature:

- First, the certificate of the sub-CA is verified by checking the signature embedded in the certificate using the public key of the CA. After this step, the public key of the sub-CA, which is embedded in the certificate, can be trusted.
- Second, the signature generated by the sub-CA can be verified using the public key of the sub-CA.

The format of certificates has been standardized for different use cases: X.509 is the standard certificate format which is, for example, used for Transport Layer Security (TLS) certificates [**? ?**]. It is very flexible and widely accepted in various industries but is not well-suited when bandwidth and memory size are limited. Certificates in the X.509 format contain a version and serial number, the identify of the issuing entity and of the owner. Furthermore, the validity period is specified and as the essential information, details on the key for which the certificate has been issued, i.e. the key itself and the algorithm for which the key shall be used, are stated. Additionally, there may be extensions of the certificate. Each extension consists of an ID, a flag which marks the extension as critical or uncritical and a value. Extensions can, for example, be used to specify for which application, e.g. signing, the certificate has been issued or if it belongs to a CA.
Card Verifiable Certificates (CVCs), which have been developed for smartcard applications and are standardized in [**?**], are an alternative which provide significant savings in view of the size of certificates. In contrast to X.509 certificates, CVCs certificates only contain the issuer, the owner, the public key and the period of validity. In particular, the certificates cannot be extended with additional information and therefore are of a limited and predictable size which is crucial in limited surroundings, such as smartcards or other embedded devices.

Chapter 4 Timing Integrity and Resource Sharing Integrity

This chapter focuses on **timing integrity** constraints associated with real-time applications, and the impact **of shared resources** on real-time constraints.

4.1 Introduction

4.1.1 Safety-Critical and Time-Critical Context

In the safety-critical domain, industries such as avionics, automotive, space, healthcare or robotics are facing an exponential growth of performance requirements in term of pure efficiency and number of embedded functionalities [**? ? ?**]. As a proxy to this exponential growth, Figure 4.1 is showing, with a logarithmic scale, the increase of embedded code size in the avionic, space and automotive industries.

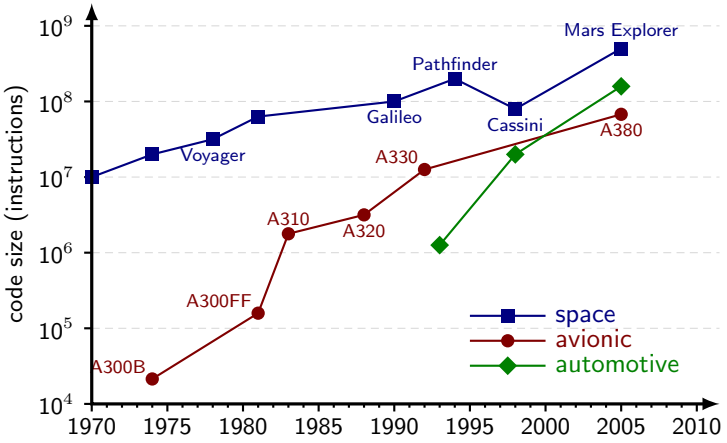


Figure 4.1: Evolution of code size (in instructions) in Space, Avionic and Automotive safety critical systems.

To cope with such performance requirements while reducing both the non-recurring engineering costs (NRE) and the Time-To-Market (TTM), these industries are relying on Commercial off-the-shelf (COTS) architectures rather than in-house solutions [**?**]. However, COTS providers are mainly targeting the consumer electronic market, mostly driven by **best-effort performances**. The safety-critical industry has to face more and more runtime variability issues [**? ?**].
However, safety-critical applications (and especially time-critical applications) are characterized by stringent real-time constraints and missing a single deadline may have catastrophic consequence on the user or the environment (such as a plane crash in avionics). Therefore, **time predictability and determinism** is a major concern, and the safety-critical industry has to respect dedicated standards [**? ? ?**] to ensure that such an unacceptable deadline-miss cannot happen.

4.1.2 Dealing with Worst-Case Execution Time

A common practice to guarantee the deadlines of a safety-critical application with single-core architecture is to determine the application Worst Case Execution Time (WCET). This WCET computation usually relies on analysis tools based on static program analysis tools [**?** **?**], detailed hardware model, as well as measurement techniques through execution or simulation [**?**]. However, these analysis techniques and tools are not currently able to provide an exact computation of the WCET, only delivering an estimated upper bound, introducing some safety margins as depicted in Figure 4.2.

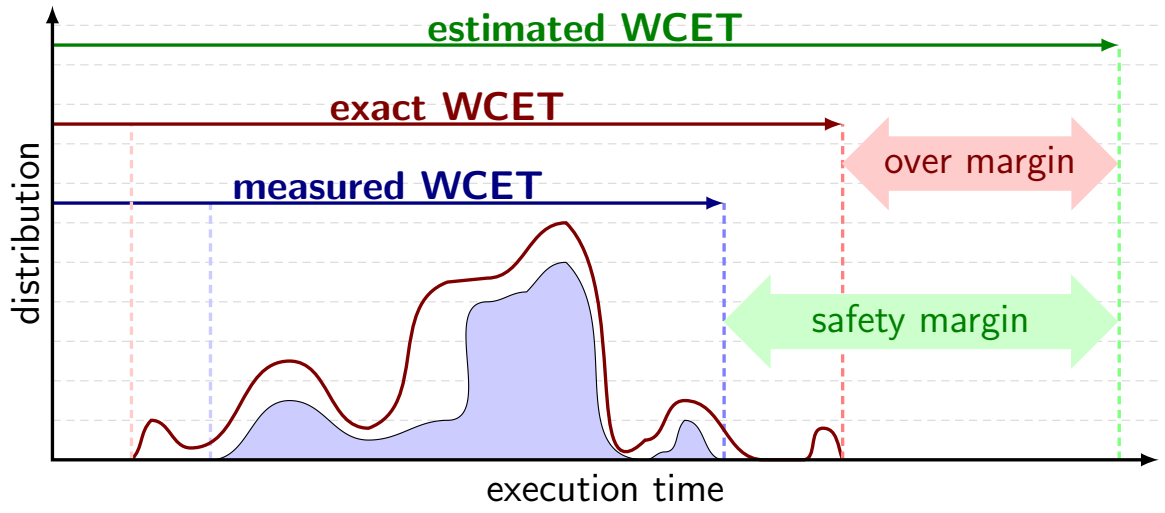


Figure 4.2: Estimation of the Worst-Case Execution Time, and the over-estimation problem

Despite all the improvements in the WCET estimation domain [**?** **?**] over the last decades, the over-estimation remained mostly constant as the predictability of the architecture decreased [**?**]. This makes the use of WCET analysis tools difficult for real industrial programs running on multi-core COTS architectures [**?** **?**].
Several studies [**?** **?**] have shown that the order of magnitude on the variation of the maximum observed execution time while using the 8 cores of a multi-core architecture was larger than the expected gain from using a multi-core (with up to a 20x on the worst case for 8 cores).
As a consequence, relying on a safety margin as usual is no longer an option, as overmargin will be far over the expected performance benefits. This leads to a worst-case performance degradation compared to single-core architectures. It is therefore critical to understand and control the sources of these variations.

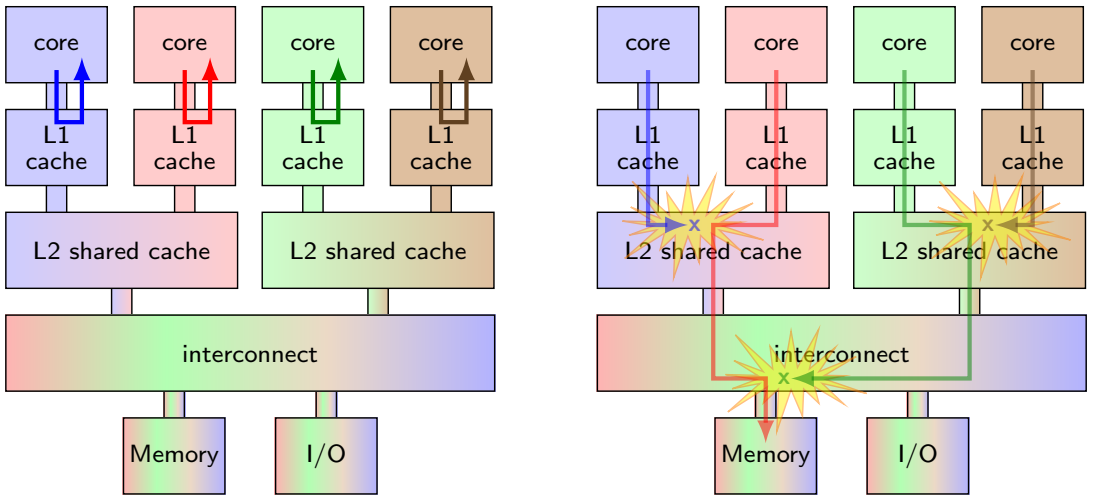
4.1.3 Multi-Core & Interferences: An Issue for Time Determinism

Multi-core architectures are potentially providing more performance than single-core architectures by allowing the parallel execution of several applications or tasks. To cope with the increase of performance-demanding functions in safety critical systems, it is becoming critical to exploit such level of parallelism. At the same time, the **spatial isolation** and **timing isolation** properties required by the safety standards must be ensured, meaning that timing integrity constraints must be met with some degree of determinism.
However, running tasks in parallel introduces some **interferences** that may endanger these properties.

4.1.3.1 Defining Interferences

Multi-core architectures are characterized at hardware-level by the fact that the different cores are sharing some hardware resources such as some caches or the interconnect of the main memory.

When several co-running tasks are executed on different cores and are trying to concurrently access the same shared hardware resource, some arbitration mechanisms are involved at hardware level. This allows one of the tasks to access the resource, while delaying all the other concurrent accesses, resulting in some contention as depicted in Figure 4.3.



(a) accesses to private hardware resources

(b) accesses to shared hardware resources ⇒ contentions

Figure 4.3: Concurrent accesses to hardware resources in a multi-core system

The tasks in Figure 4.3a exhibit no particular problems when several cores are concurrently accessing private hardware resources. However, the tasks in Figure 4.3b exhibit some contentions when several cores are requiring to access the main memory.
From a functional point of view, this behavior is not an issue: The core that has been denied access will try again during the next cycles. However, from the timing integrity point of view of each delayed task, the extra delay of a memory access, introduced because of the unpredictable behavior of other cores, are **interferences** that are breaking the time independence principles required by the standards.
The challenge is therefore to be able to ensure the timing integrity of real-time systems providing some degree of determinism to ensure a sufficient level of time independence, while dealing with time interferences.

4.1.3.2 Timing Covert Channel From Shared L2 Cache

A cache is protected from direct attacks against the confidentiality, integrity and availability of content with the help of the Memory Management Unit. However, observing the timing behavior can reveal the information on the use of the cache by other applications. This timing behavior can be leveraged to form a covert channel between the partitions. A channel is called a timing covert channel when the basis of information transfer is not by direct copying of data but rather by modulating and observing the availability or behavior of a physical or logical resource.
Assume two applications that belong to different security domains that are not supposed to exchange information are integrated in a computing platform. A Separation Kernel (i.e. a special type of operation system) ensures isolation between applications by providing temporal and spatial partitioning of system resources. If the L2 cache is shared between the partitions, one application can replace the cache line belonging to the other application by relying on the replacement policy of the cache. If the two applications are scheduled in consecutive time windows, an application can modify the state of cache to encode data and the other can decode the data from the modified cache state. The access time for reading from the cache is much lower than reading from the main memory. This difference in

access time, based on whether the memory block is cached or not, can be used to decode the data. Ristenpart [**?**] and Xu [**?**] describe this type of covert channel on a virtualized environment such as Amazon EC2. The basic algorithm used by Ristenpart [**?**] works as follows. The application that transmits the data using the covert channel is named Transmitter (TX) and the one receiving data is named Receiver (RX). In the preparation phase, the receiver makes the cache hot by bringing a memory block from its address space to the L2 cache. The L2 cache is divided into two groups based on the cache line index. The cache lines with even indexes form the even group and the cache lines with odd indexes form the odd group. To transmit a bit the transmitter evicts all lines from one group and leaves the other group untouched. I.e. to transmit a one the transmitter evicts the cache lines of Receiver indexed by odd indices, and to transmit zero, cache lines with even indices are evicted. For decoding the data the receiver reads the cache lines belonging to the even group and the odd group separately and the access time is compared. A higher access time on the odd group is decoded as one and higher access time on the even group is decoded as zero.
Kuzhiyelil et al [**?**] evaluate the L2 cache based timing covert channel on hardware (PowerPC based SoC) to identify optimum conditions for achieving maximum information flow through the covert channel with minimum transmission error. They discuss the countermeasures for timing based covert channel on the chosen hardware and real-time hypervisor PikeOS and propose a solution to mitigate and eliminate such attacks based on a proper configuration of both of hardware and OS.

4.2 State-of-the-Art on Timing Integrity for Multi-Cores

In the literature, there are two families of solutions to ensure a deterministic usage of multi-core architectures: **Control solutions** aiming at eliminating all interferences and **regulation solutions** aiming at keeping the impact of interference below a harmful level [**?**].
The literature also alternatively proposes to rely on more deterministic architectures [**? ? ? ? ? ?**]. However, these approaches rely on custom hardware and are very costly. They are beyond the scope of the SAFURE project that focuses on low-NRE COTS architecture solutions. Also such purely deterministic architectures suffer from inherent performance issues.
In the following subsections, we will present existing **control-based** and **regulation-based** software solutions to ensure time determinism with some non-deterministic COTS multi-core hardware platforms. Most of these solutions have been presented in [**?**] as **Deterministic Platform Software (DPS)** together with an evaluation for the avionic domain that is very constrained by timing integrity requirements. DPS are pieces of software that bring no functionality by themselves, but are in charge of enforcing usage domains over shared resources, effectively providing some determinism by either controlling the access to the hardware or by reactively regulating the resource usage.

4.2.1 Control Solutions Avoiding Interferences

Control solutions are usually enforcing a **usage domain** [**? ?**] that explicitly restricts the way shared resources can be concurrently used by the cores. These techniques usually rely on specifically restricted scheduling policies that ensure the absence of interferences.

4.2.1.1 Deterministic Execution Models

The Timing Integrity algorithm of the **Deterministic Execution Model** approaches [**? ? ?**] maintains the processor in a predictable state by sequentializing accesses to shared memory resources prone to interferences, while still allowing the parallel execution of non-communication sections.
More specifically the AER execution model (named after its phases), described in [**?**], distinguishes three phases: **acquisition**, during which fresh data are copied from central memory into some private memory; **execution** performing calculation on the data in isolation within the private memory; and **restitution** when results are copied back into central memory.

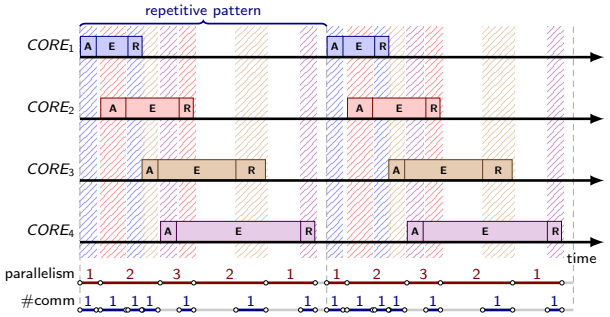


Figure 4.4: AER Execution Model

A global scheduling policy is then applied so that only one acquisition or restitution phase occurs at a given time, while not restricting the schedule of the execution phases, as shown in Figure 4.4.
A drawback of this solution is the requirement to modify the application source code to fit with the phase decomposition. Such a rewriting of the application can be costly as it requires to re-certify the application.
A major advantage is that the solution relies on a static scheduling policy, a proven technology that will make the certification of the DPS software itself straightforward.

4.2.1.2 Deterministic Adaptive Scheduling

Fischer et al. [**?**] propose the implementation of a multi-core run-time system certified for SIL-4 [**?**], particularly focusing on the railway domain.
Their algorithm, ensuring timing integrity, forces the user to define which applications (or partitions) should not suffer interferences. These applications will be run in isolation in some dedicated time slots. Non time-critical applications will - on their side - be run in parallel in interference prone time slots, as shown in Figure 4.5.

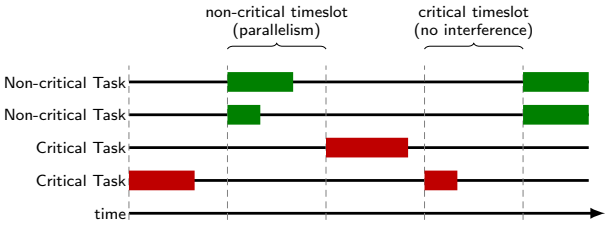


Figure 4.5: Deterministic Adaptive Scheduling

This solution can easily be applied when running critical and non-critical applications together. However, in a mixed-critical context that involves several levels of criticality, only the fully non-critical applications will benefit from parallelism, leading to a performance issue.

4.2.1.3 Marthy

Jean [**? ?**] proposes **Marthy**, a solution enabling all the cores to run critical applications, with only one of them being allowed to access the shared resources (including main memory) at a given time through a TDMA scheduling of accesses to these resources.
As for AER, sequential resource access eliminates interference, thus providing determinism. As illustrated by Figure 4.6, when a core tries to access a memory location that is not mapped in the cache, Marthy catches the associated interrupt and blocks the core execution until its assigned TDMA slot by performing low-level MMU management and cache locking.
Even though Marthy ensure strict isolation, not requiring legacy applications to be modified, it suffers two drawbacks: First Marthy is quite complex, performing low-level hardware tuning to ensure the

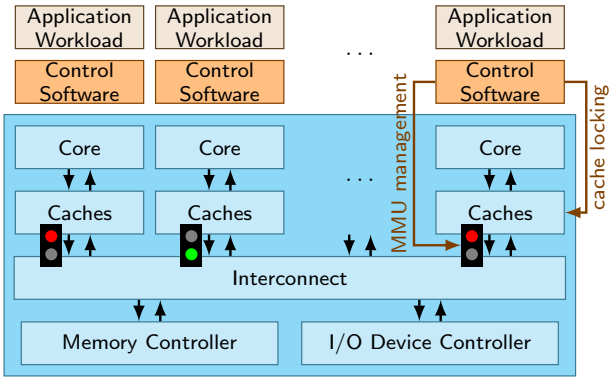


Figure 4.6: Marthy Deterministic Control Software

TDMA behavior. As a consequence, the DPS itself will be costly to certify. And second, with Marthy, the running applications are suffering a significant slowdown.

4.2.2 Regulation Solutions Keeping Interference Below a Harmful Level

While completely avoiding interferences might be necessary for most critical systems such as in avionics, most of the time making sure that interferences are not harmful is enough for less critical systems. Regulation software solutions aim at managing interferences so that critical applications will not miss their deadlines because of these interferences, effectively ensuring a sufficient level of timing integrity.
Regulation solutions have the opportunity to impact lower critical tasks to make sure that higher critical tasks are fulfilling their timing integrity requirements. As a consequence, such regulation techniques are more suited in a mixed-critical context.

4.2.2.1 Memguard

The **Memguard** time-integrity mechanism [**?**], introduced in the *Single-core Equivalent Virtual Machines* project [**?**], relies on smartly pre-allocating bandwidth to the memory system, to bound interferences.
As depicted in Figure 4.7, Memguard relies on performance counters to check the current shared bus usage versus the generated maximum bandwidth that guarantees an acceptable level of interference.

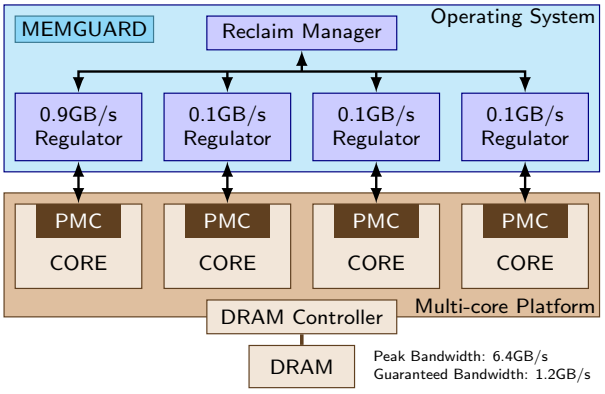


Figure 4.7: Memguard Reservation and Reclaiming System

The sum of the observed bandwidths is kept below the memory system sustainable bandwidth. When the maximum number of accesses is reached, Memguard stops the core execution until the beginning of the next time slot.

As a consequence, for each timeslot, the system integrator is responsible for allocating bandwidth to each core, making sure that the total allocated bandwidth should not exceed the maximum sustainable bandwidth.
Memguard also allows to further enhance the system performance by sharing unused bandwidth over several timeslots, thanks to a reclaim mechanism.

4.2.2.2 Distributed Runtime WCET

A last regulation software solution is proposed in [**?**] that relies on a **distributed run-time WCET controller**. This solution, developed for mixed-critical systems as part of the DREAMS project [**?**], enables to smoothly run critical tasks together with non-critical tasks.
To do so, critical tasks regularly check if the interferences due to other tasks can be tolerated. Otherwise, critical tasks request the WCET controller to temporarily suspend low criticality tasks as shown in the scenario presented in Figure 4.8.

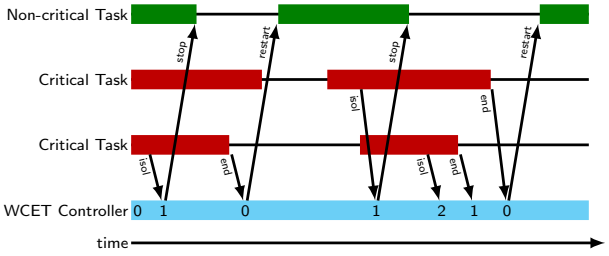


Figure 4.8: Distributed Run-time WCET Controller

A drawback of this particular solution is the necessity to modify critical applications to add the intermediate checks while the non-critical software runs unmodified. Unfortunately it means to modify the applications that are the most costly to certify.

4.2.2.3 Conclusion

Unlike control software solutions, regulation solutions are better suited in a mixed critical context, allowing to exploit the performance of the multi-core, effectively inducing some interference, but reducing their overall amount. Such regulation techniques often rely on budgeting, therefore requiring some extra analysis steps to accurately determine these budgets without pessimistic over-provisioning.

4.2.3 Multi-Core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement

In [**?**] a novel approach of computing an interference-sensitive WCET considering variable access delays due to the concurrent use of shared resources in multi-core processors, particularly focusing on shared interconnects and main memory has been proposed. This approach tackles the problem of temporal partitioning as required by safety-critical applications. The approach employs additional phases to state-of-the-art timing analysis techniques to analyze an applications resource usage and compute an interference delay. Authors further complement the offline analysis with a runtime monitoring concept to enforce resource usage guarantees. Results show a significant reduction of the multi-core WCET, while implementing full transparency to the temporal and functional behavior of applications, enabling the seamless integration of legacy applications.

4.3 State-of-the-Art on Timing Integrity for Ethernet

The formal verification of Ethernet timing integrity has received a lot of attention in recent years. In standard Ethernet (IEEE 802.1Q), the arbitration in the output ports of Ethernet switches is event-

triggered and follows Static Priority Non-Preemptive (SPNP) scheduling. The formal timing analysis for SPNP is well-understood from Controller Area Network (CAN) bus timing analysis [**?**]. In contrast to CAN, however, Ethernet offers only eight traffic classes (priority levels). Hence, different traffic streams must share priority levels. Ethernet switches typically use First In, First Out (FIFO) scheduling to arbitrate between different traffic streams with identical priority level.

4.3.1 Switched Ethernet

A timing analysis, which combines SPNP scheduling with FIFO scheduling in the context of IEEE 802.1Q is presented in [**?**]. Further analyses for IEEE 802.1Q are presented in [**?**] and [**?**].

4.3.2 ADFX - Avionics Full-Duplex Switched Ethernet

Analyses to derive worst-case timing guarantees for Avionics Full Duplex Switched Ethernet (AFDX) [**?**], an avionics Ethernet implementation which also uses SPNP scheduling, are presented in [**?**], [**?**], [**?**], [**?**], and [**?**]. Additionally, [**?**] and [**?**] present formal methods to derive buffer size bounds for switches. A holistic timing analysis for AFDX is presented in [**?**].

4.3.3 Ethernet AVB - Audio Video Bridging

Ethernet Audio Video Bridging (AVB) (IEEE 802.1Qav) [**?**] introduces standardized credit-based traffic shaping to IEEE 802.1Q to provide bandwidth-limited link access. Formal timing analyses for Ethernet AVB are presented in [**?**], [**?**], [**?**], [**?**], and [**?**]. The timing analysis in [**?**] only provides timing guarantees per traffic class, where the others allow per-stream timing guarantees. Furthermore, [**?**] shows that, while Ethernet AVB's traffic shapers can help to achieve timing integrity by limiting the impact of shaped traffic classes on lower-priority traffic, the end-to-end latency of shaped traffic classes suffers heavily from the shaper implementation.

4.3.4 Ethernet TSN - Time-Sensitive Networking

Ethernet Time-Sensitive Networking (TSN) [**?**] is a set of upcoming Ethernet standards, which, among other things, introduce new quality-of-service mechanisms to Ethernet. Among these mechanisms is the time-aware shaper (IEEE 802.1Qbv) [**?**], which introduces time-triggered slots for latency-critical traffic streams. Non- or less-latency-critical traffic streams are scheduled event-triggered outside these slots. As IEEE 802.1Qbv enforces that non-time-triggered traffic does not overlap with time-triggered slots and time-triggered traffic remains within its designated slots, it is easy to achieve timing integrity for time-triggered traffic. Its static nature, however, deems it less efficient for dynamic (i.e. event-triggered) traffic and network changes typically require a recomputation of the schedule of the time-triggered slots. Ethernet TSN also proposes two shapers for event-triggered traffic. Cyclic queueing and forwarding (IEEE 802.1Qch) [**?**] tries to bound the residence time of frames in Ethernet switches. It divides time in alternating intervals: even and odd. Frames received in an even interval are sent in the next odd one and vice versa. The burst-limiting shaper [**?**] specifies a credit-based traffic shaper similar to Ethernet AVB's shaper. However, the burst-limiting shaper allows bursts of frames up to a certain size to pass without credit replenishment. In [**?**] formal analyses for all three Ethernet TSN schedulers are presented. These analyses, however, do not consider certain blocking effects, e.g. blocking caused by FIFO scheduling among frames of identical priority is not taken into account. Hence, [**?**] cannot be used to guarantee timing integrity for Ethernet TSN.

4.3.5 TTEthernet

Similar to Ethernet TSN's time-aware shaper, TTEthernet [**?**] also provides time-triggered link access (and hence timing isolation) for latency-critical traffic streams. Timing analysis of the time-triggered

frames is, by design, trivial, once a schedule has been constructed. Additionally, TTEthernet supports event-triggered rate-constrained and best effort traffic streams.

4.3.6 Analysis Optimizations

Most formal timing analyses use certain overapproximations in order to keep their analysis complexity (and, in turn, their run-time) at an acceptable level. Typically, each resource along a chain of resources, e.g. along a path through an Ethernet network introduces a certain overestimation on the actual worst-case (timing) bounds. This can lead to overly large worst-case guarantees. In order to minimize this effect, several optimizations, which exploit different Ethernet properties, have been proposed. The analysis in [**?**] exploits that Ethernet links implicitly act as traffic shapers (e.g. 100 MBit/s) and that Ethernet AVB's traffic shaper's may further limit the bandwidth of shaped traffic classes. In [**?**] and [**?**] analyses, which exploit the FIFO scheduling in Ethernet are presented.

4.3.7 Ingress Filtering

All presented shapers have in common that they are only able to shape a certain traffic class (priority level). As already mentioned, traffic classes in Ethernet are shared among different traffic streams. This implies that these shapers can only be used to prevent or bound the inter traffic class interference, i.e. they can only be used to enforce the inter class timing integrity. In order to guarantee intra class timing integrity, per traffic stream shaping is necessary. In the context of Ethernet TSN, this is being discussed in the IEEE 802.1Qci [**?**] standard.

4.4 Vulnerability Detection for Multi-Cores

This section identifies the main hardware shared resources in the Telecom use case platform and how their sharing can affect execution time of the different tasks running. This challenges the estimation of execution time bounds, which is a form of vulnerability since safety real-time tasks need reliable execution time bounds for being properly scheduled. Therefore, a methodology is provided to upper bound the impact in execution time of contention when accessing hardware shared resources.

4.4.1 Shared Hardware Resources in the Telecom Use Case Platform

The hardware platform selected for the Telecom use case is depicted in Figure 4.9. It is composed of two quad-core clusters. The first one is a **high-performance** A57 quad-core cluster, with a 2MB L2 cache shared between the 4 cores, and private L1 caches. The second one is a **low energy** A53 quad core cluster sharing a smaller 512KB L2 cache between the cores, also encompassing private L1 caches (though being smaller than for the other cluster).
The clusters are connected to the main memory through a shared CCI400 AMBA bus and then through an Application Fabric interconnect, itself connected to a dual port 32-bit DDR controller.
These interconnect shared hardware resources are also shared by other hardware resources, like some low-level peripherals for the AMBA bus, the multi-media subsystem for the Application Fabric, as well as the System Bus connecting all the peripherals (PCI-X, UART, SPI, ...).

4.4.2 On-Chip Resource Sharing

The research on timing analysis for multi-core processors is still in its infancy. Especially so for COTS multi-cores, whose timing analysis is a complex challenge that needs to be solved before their adoption in safety-critical real-time systems industry may become viable. Deriving an Execution Time Bound (ETB)[1] for tasks running on multi-cores is challenged by the contention, also known as inter-task interference, occurring on access to hardware shared resources. Unless otherwise restrained,

[1] Due to the lack of definitive Worst-Case Execution Time (WCET) estimation methods for COTS multi-cores, we use the term "execution time bound" (ETB) instead of WCET.

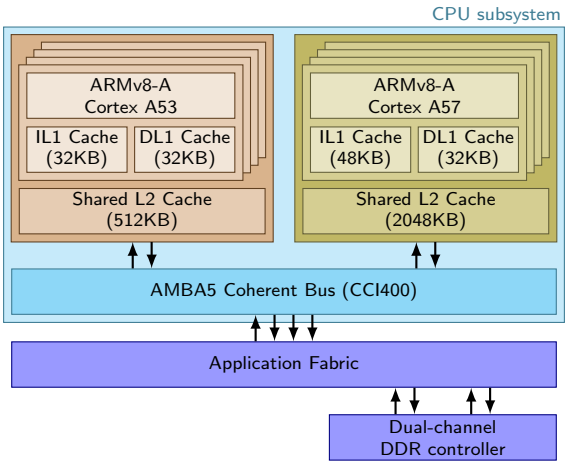


Figure 4.9: Shared Hardware resources alongside the memory path on the Telecom use case hardware platform (Dragonboard 810). As shown each of the two quad-core clusters includes a shared L2 cache and communication channels towards the DDR memory controller.

contention causes the execution time of any one task, hence its ETB, to depend on its co-runners. This has a severe impact on system design and validation, as it conflicts with the incremental development and verification model that industry pursues to contain qualification costs and development risks. This industrial goal is solved by allowing individual subsystems to be developed in parallel, qualified in isolation, and incrementally integrated, with great reduction in the risk of functional regression at system level. In the timing domain, incremental integration and qualification postulate *composability* in the timing behavior of individual parts, whereby the ETB derived for a task determined in isolation, should not change on composition with other tasks.

Two main approaches have been followed so far to deal with contention for multi-core on-chip resources, which place at the opposite ends of a conceptual spectrum of solutions. On the one end, some authors propose computing ETBs so that they upper bound the effect of any possible inter-task interference a task may suffer on access to hardware shared resources. ETBs computed this way are *fully time composable* and enable incremental integration and qualification, but at the cost of pessimism that may cause untenable over-provisioning, as the timing behavior occurring in operation may fall much below the level determined considering the worst-case interference possible in theory [**?** **?** **?**]. On the opposite end, other authors [**?**] propose – currently only for research platforms – to determine ETBs simultaneously for multiple tasks in specific configurations. Those ETBs are *non-time composable*, as they only hold valid for the tasks being analyzed and for their specific configuration. If any such parameter changes, all ETBs become invalid and the entire analysis has to be repeated.

In this study we tackle resource contention in multi-cores by proposing the new concepts of **resource usage signature** (*RUs* or $\mathcal{S}$) **and template** (*RUl* or $\mathcal{L}$). A task τ is exposed to contention in the access to the hardware shared resources because co-runner tasks can interfere τ's execution. The usage of the hardware shared resources made by co-runner tasks is referred to as u. *RUs* and *RUl* aim at making the ETB derived for τ, time composable with respect to u. The tasks' ETBs are derived for a particular set of utilizations $\mathcal{U}$ such that the ETB derived for any $u \in \mathcal{U}$ upper bounds τ's execution time under any workload so long as the co-runners of τ make a resource usage smaller than u. We explain later what "smaller" means and how this can be determined. This abstraction allows deriving time-composable ETBs for individual tasks in isolation for each $u \in \mathcal{U}$, so that the system integrator can safely pull those (interfering) tasks together as long as the resource usage made by their individual set of co-runners is upper-bounded by some u. All that the system integrator has to care in that regard is to characterize the tasks' accesses to hardware shared resources (a low-cost abstraction of the task execution time), ignoring any finer-grained detail of that access behavior. In this section we present an approach to produce ETBs in that manner, using measurement-based

timing analysis techniques.
RUs and *RUI* are devised to be agnostic to the particular timing distribution of the resource access requests to be considered. Hence, two tasks generating the same number of accesses to a resource, though with different patterns, have the same signature. The challenge in the proposed method is in determining an effect on the interfered task that upper bounds the interference caused by contending accesses, regardless of the time distribution of those accesses as made by the interfered and the interfering tasks. In this work we make the following main contributions:

1. We develop the novel concepts of *RUs* and *RUI* for the timing analysis of COTS multi-cores and sketch an algebra of operators over *RUs*/*RUI* to enable their practical use.
2. We provide exemplary *RUs* and *RUI* for the cases when requests accessing shared resources incur either fixed or variable response latency.
3. We present a strategy to implement *RUs* and *RUI* for the SnapDragon processor used in the Telecommunications case study [**?**] architecture, focusing on the bus and the memory controller as exemplars of on-chip shared resources.

4.4.2.1 Formalization of RUs and RUI

RUs and *RUI* allow analyzing, for the most part in isolation, the timing behavior of tasks, by abstracting the interference that they may suffer due to contention when accessing hardware shared resources on a multi-core caused by co-runner tasks.

4.4.2.1.1 Resource usage signature (*RUs*)

Given an interfered task, τ_A, a *RUs* abstracts its use of the hardware shared resources. Once computed, it will be used for τ_A's multi-core timing analysis instead of τ_A itself.
We describe the use of a hardware shared resource through a set of features, which correspond to quantitative values. A *RUs* for task τ_A, is a vector $\mathcal{S}_A = (a_1, a_2, ..., a_n)$ that contains the list of relevant features that characterize all the hardware shared resources, for the evaluation of contention effects. Since *RUs* are quantitative, the *RUs* of distinct tasks are comparable and can also be combined together to form a joint *RUs*. How to operate and compare multiple *RUs* is explained later.
Consider the reference multi-core architecture shown in Figure 4.10, where the bus and the memory are shared. Further consider two types of accesses to those shared resources, for read and write operations respectively. In this case, *RUs* have at most 4 features: bus reads (n_{rd}^{bus}) and writes (n_{wr}^{bus}); memory reads (n_{rd}^{mem}) and writes (n_{wr}^{mem}). *RUs* are thus defined as $\mathcal{S}_A = (n_{rd}^{bus}, n_{wr}^{bus}, n_{rd}^{mem}, n_{wr}^{mem}) = (a_1, a_2, a_3, a_4)$.
If the bus was the only shared resource, the *RUs* of a task τ_A would be abstracted as a *RUs* with two features: n_{rd}^{bus} and n_{wr}^{bus}. If both types of requests hold the bus for the same duration, the *RUs* would consist of a single feature corresponding to the sum of n_{rd}^{bus} and n_{wr}^{bus}, i.e., $\mathcal{S}_A = (n_{rd}^{bus} + n_{wr}^{bus}) = (a_1 + a_2)$. The addition of $\mathcal{S}_B$ to $\mathcal{S}_A$ is given by $\mathcal{S}_A + \mathcal{S}_B = (a_1 + a_2 + b_1 + b_2)$. For comparison, instead, we say that $\mathcal{S}_A$ dominates $\mathcal{S}_B$, $\mathcal{S}_A \gtrsim \mathcal{S}_B$, if the interference by the former is greater than that by the latter: $a_1 + a_2 \geq b_1 + b_2$.
This reasoning easily extends to the more realistic scenario in which the bus holding times are asymmetric; for example, with reads holding the bus longer than writes. In that case, the *RUs* for τ_A could be either single-feature, considering all accesses as "long" accesses (counting writes as reads in the example), or multi-feature (two, in the example), i.e., $\mathcal{S}_A = (a_1, a_2) = (n_{rd}^{bus}, n_{wr}^{bus})$. In the latter formulation, addition and comparison change as follows: addition is defined as vector addition, i.e., $\mathcal{S}_A + \mathcal{S}_B = (a_1 + b_1, a_2 + b_2)$; for comparison, $\mathcal{S}_A$ dominates $\mathcal{S}_B$, $\mathcal{S}_A \gtrsim \mathcal{S}_B$ if $(a_1 \geq b_1) \wedge (a_2 \geq b_2)$.
Note that it can be the case that neither $\mathcal{S}_A$ dominates $\mathcal{S}_B$ nor $\mathcal{S}_B$ dominates $\mathcal{S}_A$ if, for instance, $(a_1 > b_1) \wedge (a_2 < b_2)$. In that case one should resort either to single-feature *RUs* or to having a new multi-feature *RUs* $\mathcal{S}_C = (max(a_1, b_1), max(a_2, b_2))$ so that $\mathcal{S}_C$ dominates both $\mathcal{S}_A$ and $\mathcal{S}_B$.

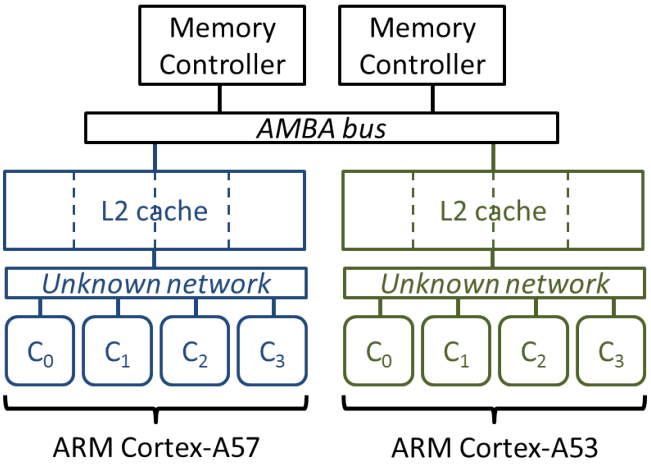


Figure 4.10: Reference multi-core architecture.

4.4.2.1.2 Resource usage template (*RUI*)

RUI have the same form as *RUs*, namely, a vector of features $\mathcal{L}_K = (k_1, k_2, ...k_n)$, but with a different use. *RUs* abstracts tasks according to their use of the shared resources while *RUI* upper bounds the use of the shared resources made by co-runner tasks. A *RUI* is a true upper bound if $\mathcal{L}_K \gtrsim \mathcal{S}_i$, where $\mathcal{S}_i$ is the *RUs* for any task τ_i that can be a co-runner of the task under analysis (i.e. $\mathcal{S}_i$ is dominated by $\mathcal{L}_K$).
Tasks are made time composable against some *RUI* $\mathcal{L}_K$ so that the ETB derived for a given task τ_A and for that *RUI*, denoted ETB_A^K, upper bounds τ_A's execution time inclusive of the interference that the contenders of τ_A, whose *RUs* do not exceed $\mathcal{L}_K$, may cause.
Returning to the example in which the bus is the sole shared resource with all accesses to it incurring the same contention effect: for a $\mathcal{L}_K$ that captures a given number of accesses to the shared bus, we want to determine the highest impact by $\mathcal{L}_K$ on ETB_A, so that ETB_A^K can be regarded as a time-composable bound for τ_A in any workload in which $\mathcal{L}_K \gtrsim \sum_i \mathcal{S}_i$ for all co-runner tasks τ_i of interest.
A maximally time-composable template $\mathcal{L}_{TC}$ exists, which is an upper bound for all potential workloads (i.e. including the worst possible workload). $\mathcal{L}_{TC}$ corresponds to the case in which all accesses from the signature suffer the highest contention from the $N_c - 1$ contending cores[2]. In that case, every access from $\mathcal{S}_A$ contends with $N_c - 1$ other accesses, i.e., $\mathcal{L}_{TC} = (N_c - 1) \times \mathcal{S}_A$. Any $\mathcal{L}_K \gtrsim \mathcal{L}_{TC}$ would produce exactly the same result as $\mathcal{L}_{TC}$, since τ_A cannot be interfered more than the accesses in its signature $\mathcal{S}_A$.

4.4.2.1.3 RUs and RUI through an example

In this section we return to the case in which the bus is the sole shared resource and all accesses to it incur the same contention effect. For now we limit our attention to two cores. The task under analysis, τ_A, runs in one of the two cores. The contending requests from the two cores are arbitrated with the round-robin policy.
Figure 4.11 depicts the process we follow when the proposed approach is applied to this case. First, we obtain the *RUs* of τ_A, denoted $\mathcal{S}_A$. In the example architecture, the *RUs* of tasks using the shared resource is the number of accesses they make, a for τ_A, hence $\mathcal{S}_A = (a)$. Our approach treats

[2]Note that the highest contention for a given access can be upper-bounded in modern embedded processors [**? ? ?**], so such highest contention can be computed in general. If, for instance, one core had priority over the others on the access to hardware shared resources, then the highest contention possible would be infinite and an ETB would not exist. Then such hardware platform would not allow running any critical task in the low-priority cores and our analysis would be valid only for the high-priority cores.

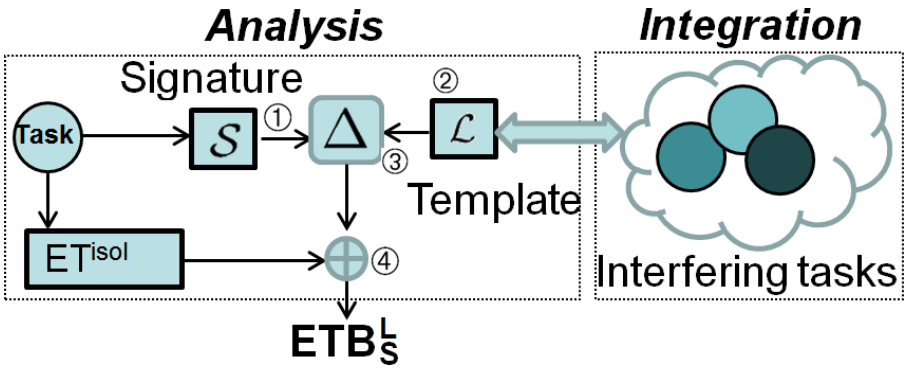


Figure 4.11: Main steps in the *RUs* and *RUI* methodology.

contention such that the ETB of τ_A can be derived by upper bounding τ_A's execution time considering the interfering effect that it incurs when its co-runner task makes up to k contending accesses to the shared resource. To this end we define a *RUI* $\mathcal{L}_K$, which is the system integration parameter that defines the inter-task interference to be considered in the determination of τ_A's ETB. The abstraction captured by $\mathcal{L}_K$ with $\mathcal{L}_K = (k)$ is a *RUI*.

Once the $\mathcal{S}_A$ and $\mathcal{L}_K$ are defined, we determine Δ_A^K, the increment to be applied to the execution time that τ_A may incur, to capture the contention effect from $\mathcal{L}_K$. This corresponds to step 3 in Figure 4.11. More precisely, Δ_A^K upper bounds the increment that the execution time of a task τ_A with at most a accesses to a shared resource may suffer from k contending requests. ETB_A^K (i.e τ_A's ETB determined under the *RUI* $\mathcal{L}_K$) is computed as the summation of ET_A^{isol}, the execution time of τ_A when running in isolation, without contention, and Δ_A^K, the increment that upper bounds the contention effects from any k interfering accesses. This corresponds to step 4 in Figure 4.11. Overall, ETB_A^K is time composable against any co-runner task τ_B with signature $\mathcal{S}_B = (b)$, as long as $\mathcal{L}_K$ dominates the *RUs* of the co-runner, which means that τ_B makes $b \leq k$ contending accesses. We denote this as $tc(ETB_A^K, \tau_B)$, which holds if $b \leq k$.

RUs abstract the distribution of requests over time. Taking into account the exact distribution of requests over time, for instance in the form of requests arrival curves [**?**], would potentially enable deriving tighter ETB. However, deriving such distributions is complex, as programs normally have multiple paths of execution, each with its own access pattern (distribution). And, paradoxically, considering these particular distributions would decrease timing composability. Instead, our approach only requires the tasks' access count for every individual shared resource, as well as ET_i^{isol} (execution time in isolation) for each individual task τ_i. Notably, both can be obtained with high accuracy by state-of-the-art technology, e.g., [**?**]. With our approach, the ability to abstract away from the need to know the exact points in time at which requests would be made to shared resources releases the system integrator from the obligation of adopting rigid and inflexible scheduling decisions (which fares poorly with the development unknowns of novel systems) or from the labour-intensive cost of exact analysis.

Our approach requires the user to set the *RUI* to capture the potential co-runner tasks precisely. The spectrum of this capture has two ends. On one extreme we find the time-composable templates, $\mathcal{L}_{TC}$, which represent an upper bound for *RUI*. However, if *RUI* is close to that template, the ETB of tasks might be unnecessarily increased. On the opposite extreme, if *RUI* is too small, it constrains the choice of tasks that may be allowed to run in parallel. A simple solution consists in deriving for each task an ETB under different *RUI*, such that at integration time, the smallest *RUI* that upper bounds the signature of the actual co-runner tasks is used. With this, the residual part of the timing verification at system integration is small and simple. Selecting the proper number of *RUI* represents a trade-off between effort and accuracy: the higher the number of *RUI* the lower the over-estimation of ETB and the greater the analysis time, and vice-versa. Finding appropriate *RUI* is a standard optimization problem that is part of our future work.

In the example considered in this section we have made several simplifications to facilitate understanding: two cores, one single type of access, synchronous accesses (i.e. the core stalls when the access occurs until served) and a single shared resource. In real processors we have different types of accesses to the shared resource (synchronous and asynchronous), each with a distinct access latency. Hence, simply bounding the effect of contention by adding access counts is not enough.

4.4.2.2 RUs & RUI for Measurement-Based Timing Analysis

Next we present one concrete realization of *RUs* and *RUI* for use with measurement-based timing analysis, specifically for a SnapDragon-like processor architecture [**?**].

4.4.2.2.1 Methodology

Our approach uses *micro-benchmarks* [**?** **?** **?**], a set of single-phase user-level programs with a single execution behavior designed so that all their operations access a given shared resource, e.g. the bus. Micro-benchmarks consist of a main loop whose body includes a substantial number (e.g. 256) of instructions designed to generate a steady stress load on target resources. The fact that the loop body executes repeatedly the same instruction causes the target resource to be continuously accessed. Moreover, placing a high number of identical instructions in the loop body drastically reduces the impact of control instructions (down to 2-4%) [**?**]. For the architecture in Figure 4.10, a loop body including load instructions that hit in the L2 cache stresses the bus. We consider two types of micro-benchmarks:
Resource stressing benchmarks, *RStB*, place a configurable load on a given shared resource, so that running a task against a *RStB* may represent contention scenarios of interest.
In theory, one could design a *worst-contender* benchmark that generates the maximum contention that a task τ_i can suffer. However, such benchmark would be specific for the task to be interfered and for the target processor [**?**]. Consider for example, a single shared resource arbitrated by a least-recently-used policy, where the task that accessed the resource last gets the least priority. In that case, the worst-contender benchmark should generate a request in exactly the same cycle as the task of interest, so that every request from that task gets delayed by the contender, and for the next round of arbitration the task has the lowest priority again. The level of control required on the application behavior and the granularity of intervention are too fine-grained and laborious to be used in practice [**?**].
Resource sensitive benchmarks, *RSeB*, are designed to upper bound the execution time increase suffered by any other task, with a smaller or equal signature, owing to the interference from a given template $\mathcal{L}_K$. Consider a scenario in which bus accesses hold the bus for a constant duration. Further assume that we want to determine Δ_A^K for τ_A, i.e its ETB increment due to a template $\mathcal{L}_K$ with k accesses. Intuitively, one could get an estimate of it by running τ_A several times against a *RStB* that makes k accesses. However, in order to gain confidence in the ETB obtained, the experiment should be repeated with different *alignments* of the *RStB*, so that the interleaving of accesses varies enough and the worst case can be observed in a measurement. In practice, this may require excessive experimentation effort. The need for repeating the experiments with different alignments stems from the uncertainty on the time distribution of accesses, which is hard, if at all possible, to measure and control by timing analysis technology. We can therefore conclude that studying the task under analysis against micro-benchmarks is not viable. Instead, we use micro-benchmarks *to model both the interfered and the (set of) interfering tasks*: *RStB* and *RSeB* are designed to account for bad alignments of requests: *RSeB* is made of instructions that cause accesses to the shared resource and that continuously contend with *RStB* requests.
We define $\Delta_{RSeB}^{RStB} = ET_{RSeB}^{RStB} - ET_{RSeB}^{isol}$, where ET_{RSeB}^{RStB} is the execution time when a given *RSeB* with the same signature as task τ_A runs against a *RStB* implementing a template $\mathcal{L}_K$ with k accesses; and ET_{RSeB}^{isol} the execution time when the *RSeB* runs in isolation. For task τ_A, let $\Delta_A^K = ET_A^K - ET_A^{isol}$ be the execution time increase τ_A suffers when it runs against $\mathcal{L}_K$. *RSeB* and *RStB* are designed so that $\Delta_{RSeB}^{RStB} \geq \Delta_A^K$ holds for any request alignment of τ_A under $\mathcal{L}_K$ contention. To that end, we

run the *RSeB* in isolation and then against $N_c - 1$ copies of *RStB* so that all *RSeB*'s accesses to the shared resource suffer high contention, causing a measurable Δ^{RStB}_{RSeB} to emerge. In the next section we show how to derive the number of accesses of the *RSeB* and the *RStB*, based on the number of accesses of the template and signature under consideration.
Δ^{RStB}_{RSeB} is used to compute the ETB estimate for τ_A as follows: $ETB^K_A = ET^{isol}_A + \Delta^{RStB}_{RSeB}$. ETB^K_A is composable with any set of interfering tasks against which τ_A runs in parallel, if their total number of accesses is lower or equal to k. That is, the addition of the signatures of the interfering tasks is dominated by $\mathcal{L}_K$: $(S_i + S_j + ... + S_l) \precsim \mathcal{L}_K$. Interestingly, given a task τ_B whose signature is dominated by τ_A, i.e. $S_B \precsim S_A$, the obtained Δ^{RStB}_{RSeB} for τ_A can be used to upper bound τ_B's execution time: $ETB^K_B = ET^{isol}_B + \Delta^{RStB}_{RSeB}$.
Overall, *RUs* and *RUI* provide powerful abstractions for the interfered and the interfering tasks, which simplify the integration of multiple tasks by combining their signatures.

4.4.2.2.2 The case of a SnapDragon-like architecture

Our reference multi-core architecture [**?**] comprises $N_c = 4$ symmetric cores in each of the 2 multicores (ARM Cortex-A57 and ARM Cortex-A53), see Figure 4.10, each equipped with private instruction cache (IC) and data cache (DC). Some of the hardware characteristics described next are derived from the manuals while others are inferred from limited information in those manuals. Those that may affect the methodology and cannot be obtained unambiguously from the processor manual will be studied empirically as part of BSC work in T4.1.

Relevant hardware characteristics
ARM Cortex-A57 and ARM Cortex-A53 multi-cores differ. However, the ARM Cortex-A57 is more powerful than the Cortex-A53 one, so the features of the Cortex-A57 are a superset of those in the Cortex-A53. Hence, we refer to the particular hardware characteristics of the Cortex-A57 multi-core given that they already include those of the Cortex-A53 one. IC and DC can be used in many different ways including write-allocate, write-no-allocate, write-through, etc. This has a direct influence on the number of accesses issued to the L2 cache, and so on the contention suffered. Analogously, speculation on branches and prefetch operations may issue further requests to the shared resources that may also suffer contention (or increase the contention suffered by non-speculative requests). The L2 cache can process up to 2 requests simultaneously if they access different tag/data Random Access Memory (RAM) arrays. L2 is accessed through an Advanced Microcontroller Bus Architecture (AMBA) AXI interface which, in principle, may receive requests from the different cores and process them in parallel. Therefore, it is the L2 cache the resource serializing requests, not the AMBA AXI interface.
It is, therefore, unclear what type of network is implemented. We will assume that core-to-L2 networks are also implemented using AMBA buses, as it is the case for the L2-to-memory network, since this is the worst case because requests are fully serialized and so contention is the highest. As the work in the project progresses we will confirm or reject our assumption. However, if it is rejected, our methodology should be flexible enough to adapt to any common network available in small multi-cores such as the 4-core ARM Cortex-A57 and Cortex-A53.

Bus
For the sake of this discussion we assume that our target processor implements round-robin bus arbitration so that if, in a given round, core $c_i, i \in \{1, .., N_c\}$ is granted access to the bus, the priority ordering in the next round is: $c_{i+1}, c_{i+2}, ..., c_{N_c}, c_1, c_2, ..., c_i$. A lower priority core can use the bus when all higher priority cores do not use it. Whether this is the policy in place or not will be investigated. Currently, as part of the work in T4.1, BSC is developing a method that, among others, is able to identify whether the arbitration policy in shared resources is round-robin or FIFO.
The *bus access jitter* that a task incurs on access to the bus, depends not only on the number of co-runners but also on the way their requests interleave. The worst contention situation happens when a task τ_B assigned to core c_i requests the bus in a given round of arbitration, simultaneously

interfered	st			l2h			l2m		
interfering	l2h	l2m	st	l2h	l2m	st	l2h	l2m	st
Impact	7	2	2	7	2	2	2x7=14	2x2=4	2x2=4

Figure 4.12: Hypothetical impact (in cycles) from/to the different access types to the bus. $l2h$, $l2m$ and st refer to L2 load hits, L2 load misses and stores respectively.

with tasks in all other cores and the previous round was assigned to c_i.

L2 cache
In this discussion we assume that the L2 cache processes up to one miss per core at a time and allows hit-under-miss and miss-under-miss so that when a miss from a core is processed, hit/miss requests from other cores can be served. In practice the L2 cache might process multiple requests from the same core since any core may have several pending requests. However, although it needs to be validated empirically, we expect a round-robin arbitration policy across cores, thus limiting the impact of other cores on a particular request to a single request per core, so in line with our assumption. Still, this needs to be corroborated empirically to find reliable upper bounds to the contention across cores when accessing the L2 cache. To the best of our knowledge, the L2 cache cannot be configured to be partitioned across the different cores to remove L2 cache interference. This brings a new challenge in terms of on-chip interference. Solutions reported in this deliverable neglect L2 cache interferences on purpose, since this work is to be developed in T3.3 up until M24 to complete the management of on-chip timing interferences. At this stage we plan to build upon some form of probabilistic/statistical analysis of L2 cache interferences based on the fact that L2 caches implement random replacement policies to upper-bound the number of misses that a task could suffer due to interferences and upper-bound also the impact that this may have on the number of memory accesses.

Memory controller
The L2 sends a request to a memory controller on every L2 miss. Requests are stored in a FIFO request queue, with one entry per core. Whether only one memory controller is in place or each multi-core has a separate one needs to be investigated. For the sake of this discussion we will assume that each multi-core uses one different memory controller. In future enhancements of our methodology we will relax this constraint.

4.4.2.2.3 Bus

The AMBA bus handles three distinct request types, which differ in the contention they induce and suffer. Stores (*st*) regardless of whether they hit or miss on the L2, are served immediately by the L2 and hold the bus for few cycles (e.g. 2 cycles). L2 load hits (*l2h*) hold the bus for few more cycles (e.g., 7 cycles) because the bus is retained while retrieving the data from L2. L2 load misses (*l2m*) release the bus once the request reaches L2, and perform a new arbitration whenever the L2 responds to the miss, holding the bus for as many cycles as L2 store accesses (e.g., 2 cycles) in each arbitration. Figure 4.12 shows the contention suffered by a source (interfered) request by another (interfering) request for all request types assuming some hypothetical latencies of 2 cycles for short L2 stores and L2 load misses, and 7 cycles for L2 load hits. $l2h$ generate the highest contention and $l2m$ are the most affected since they suffer two rounds of arbitration: $l2m$ can therefore be interfered twice by two concurrent contending requests, one round of arbitration per each such request.
Our approach based on *RUs* and *RUl* does not require knowing the exact time of request issue, but whether they have asymmetric timing behavior in the impact they suffer and they cause to other request types so that *RStB* and *RSeB* can be designed with the appropriate request types. The *RStB* and *RSeB* for the bus are called *BStB* and *BSeB*:

BSeB (abstracting interfered task bus usage). The signature of a task τ_A running in this archi-

tecture may take different forms, with different levels of tightness and experimentation effort. The canonical signature for the bus contains the number of accesses of each type made by the task. That is: $\mathcal{S}_A^{bus} = (a_{st}, a_{l2h}, a_{l2m})$. This can be simplified by realizing that $l2h$ and st access the bus once whereas $l2m$ do it twice. Moreover, the delay suffered by an access does not vary whether the access was generated by a $l2h$, st or $l2m$. Hence, signatures have the form: $\mathcal{S}_A^{bus} = (a_{st} + a_{l2h} + 2 \times a_{l2m})$. *BSeB* can be implemented with either $l2h$ or st. Conversely, $l2m$ are not appropriate as it is not possible to place high pressure on the bus with $l2m$ since they miss in cache and take long to be served from memory, leaving the bus idle in the meantime. Instead, $l2h$ and st can place very high pressure on the bus. Our approach considers *BSeB* to only have st operations.

BStB (abstracting interfering task(s) bus usage). Templates can be mono- ($\mathcal{L}_{1D}$) or bi-dimensional ($\mathcal{L}_{2D}$).

$\mathcal{L}_{2D}$. Accesses of type st and $l2h$ generate different impact on the bus (recall that $l2m$ are equated to 2 st). In particular, $l2h$ produces the highest impact and st the lowest. This allows generating bi-dimensional templates: $\mathcal{L}_{2D} = (k_{l2h}, k_{2 \times l2m + st})$, whereby *BStB*s comprises load L2 hit accesses and store accesses to generate each respective type of interference.

$\mathcal{L}_{1D}$ templates comprise only $l2h$, which generate the highest interference. A given $\mathcal{L}_{1D} = (k_{l2h})$ with k $l2h$ accesses upper bounds the impact that one or several tasks, whose bus access count is lesser or equal to k, can generate on any other interfered task. $\mathcal{L}_{1D}$ are easier to generate and simplify experimentation, but they increase the pessimism of ETBs, since st are considered to generate the same impact as $l2h$.

Putting it all together. Deriving the access count for *BSeB* and *BStB* varies for $\mathcal{L}_{1D}$ or $\mathcal{L}_{2D}$ as we show next.

$\mathcal{S}_A - \mathcal{L}_{1D}$. Let a and k be the number of accesses in the signature $\mathcal{S}_A$ and the template $\mathcal{L}_K$ respectively. Running *BSeB* and *BStB* concurrently, we derive an upper bound to the increase in execution time (Δ_{BSeB}^{BStB}) that k accesses of the template can have on the a accesses of the signature. If $k \geq (N_c - 1) \times a$ then each request of $\mathcal{S}_A$ suffers the impact of $N_c - 1$ contenting requests. If this is not the case, only $\lceil k/(N_c - 1) \rceil$ requests from $\mathcal{S}_A$ suffer the impact of $N_c - 1$ contenting requests.
The number of request accesses generated by the *BSeB* is given by $N = min(a, \lceil k/(N_c - 1) \rceil)$. By running this *BSeB* against $N_c - 1$ *BStB* copies, each having a number of accesses largely above N, we derive an upper bound to the impact that $\mathcal{L}_K$ has on $\mathcal{S}_A$. The impact that a task can suffer due to a template $\mathcal{L}_K$ with k $l2h$ is upper bounded as: $\Delta_{BSeB}^{BStB} = ET_{BSeB}^{BStB} - ET_{BSeB}^{isol}$. The ETB derived for a given task τ_A and template $\mathcal{L}_K$ is: $ETB_A^K = ET_A^{isol} + \Delta_{BSeB}^{BStB}$.

$\mathcal{S}_A - \mathcal{L}_{2D}$. In the case of 2-dimensional signatures and templates we account for the fact that requests sent by the interfered task, τ_A, suffer different interference by the $l2h$ and $l2m/st$ sent by the interfering tasks, abstracted in $\mathcal{L}_{2D}$. In this approach we pair up every request in τ_A with $N_c - 1$ requests in $\mathcal{L}_{2D}$ causing the highest interference ($l2h$) on the former. If the number of those requests in $\mathcal{L}_{2D}$ is exhausted, we pair up τ_A requests with those in $\mathcal{L}_{2D}$ causing the second worst interference (st).
We generate two *BSeB* and *BStB* pairs to capture the impact that accesses in $\mathcal{S}_A$ suffer from $l2h$ and $l2m/st$ in $\mathcal{L}_{2D}$ so that:

$$\Delta_{BSeB}^{BStB} = \left(\Delta_{BSeB_1}^{BStB_l} + \Delta_{BSeB_2}^{BStB_s} \right) \tag{4.1}$$

$BSeB_1/BStB_l$ and $BSeB_2/BStB_s$ capture the interference on τ_A's accesses caused by the $l2h$ and $l2m/st$ in $\mathcal{L}_{2D}$ respectively. $BSeB_1$ and $BSeB_2$ have different number of st operations, N_1 and N_2. $BStB_l$ comprises $l2h$ operations whereas $BStB_s$ comprises st operations.
Let us assume for example $a = 30$, $k_{l2h} = 60$, and $k_{st} = 80$. In this case, $BSeB_1$ has $N_1 = min(30, \lceil 60/3 \rceil) = 20$ st, which we pair up with 20 accesses in $\mathcal{S}_A$; and $BSeB_2$ has the rest of accesses in $\mathcal{S}_A$, $N_2 = 30 - 20 = 10$ st, which we pair up with 3×10 requests out of the 80 accesses in k_{st}. The remaining 50 st in k_{st} are not paired since they will not cause further impact on $\mathcal{S}_A$. Overall, an upper bound to the impact that an application can suffer due to $\mathcal{L}_{2D}$ is given by:

$$ETB_A^K = ET_A^{isol} + \left(\Delta_{BSeB_1}^{BStB_l} + \Delta_{BSeB_2}^{BStB_s}\right) \quad (4.2)$$

For the memory controller we follow the same principles as for the bus, with the particularity that the impact from/to the read/write request types is homogeneous. Hence we only need $\mathcal{L}_{1D}$ templates. The *RStB* and *RSeB* for the memory are called *MStB* and *MSeB*.

4.4.2.2.4 Multi-resource signatures

In the presence of several shared resources, the signatures and templates must cover the features to upper bound contention in each of them. For the reference architecture considered in this work, signatures and templates are as follows: $S_A^{bus+mc} = (a_{st} + a_{l2h} + 2a_{l2m}, a_{mem})$ and $\mathcal{L}_K^{bus+mc} = (k_{st} + 2k_{l2m}, k_{l2h}, k_{mem})$.
It is possible that a task suffers contention in several shared resources simultaneously, so that the impact of the contention does not accumulate but rather overlaps. However, determining trustworthy bounds to the degree of overlap in the contention suffered on requests to different resources is hard. Signatures and templates are intentionally made agnostic to the distribution of requests over time. As we focus on the number of requests to each resource rather than on their timing, it is difficult to determine how contending requests overlap. Our current approach assumes no overlap in contention, which in a time-anomaly free processor design [**?**] is a safe assumption on the maximum impact of contention. It needs to be investigated to what extent the SnapDragon processor is subject to timing anomalies and to what extent those timing anomalies break our assumptions. Overall, in the presence of a template for the bus $\mathcal{L}^{bus}$ and the memory $\mathcal{L}^{mc}$ (a.k.a. $\mathcal{L}^{bus+mc}$), a task is assumed to suffer the sum of the contention generated by both templates:
$ETB_A^{\mathcal{L}_K^{bus}+\mathcal{L}_K^{mc}} = ET_A^{isol} + \Delta_{BSeB}^{BStB} + \Delta_{MSeB}^{MStB}$

4.5 Vulnerability Detection for Networks

This section describes a timing analysis approach for networks, which can be used to detect timing vulnerabilities (usually at design time). In principle, the approach works on a timing model of the system, in which the key timing parameters are captured (see Section 5.1.1 of Deliverable D1.3). On this, different analyses can be performed: A simulation-based system distribution analysis computes the ”average” timing behavior, while a formal worst-case analysis computes the timing corner cases (see Section 5.1.2 of Deliverable D1.3).
In this section, we will describe a worst-case analysis for Ethernet networks, that has been developed and extended in the SAFURE project. With it, upper bounds on the timing of Ethernet networks can be computed, which can be used to verify timing requirements and guarantees. This way, it can be assured that a given configuration meets its timing integrity goals. In this analysis, unknown ”attacker” traffic can be captured by assuming a worst possible attack scenario (e.g. flooding of the network at a certain priority level with the maximum possible bandwidth) to derive the effects of such attacks on the timing of critical traffic.

4.5.1 Worst-Case Ethernet Analysis in SymTA/S

The worst-case analysis of Ethernet implemented in SymTA/S is based on the theory of compositional performance analysis [**?**] and the corresponding extensions from TUBS [**?**].
The analysis is based on an Ethernet model, comprising the topology (switches, ports, links, end nodes) and traffic configuration (Ethernet messages) In this context, an Ethernet message is defined by a logically independent communication stream between a sender and (one or multiple) recipient(s). For each Ethernet message, the payload size (as an interval between minimum and maximum size) and activation model (e.g. period and jitter) must be defined, as well as some additional parameters (e.g. protocol, message priority).

As a first step, the analysis computes the loads of ports on switches and end nodes based on the network configuration and traffic description in the model, using especially the size and transmission activation patterns of Ethernet messages. This is done per individual message and then accumulated for each port. The load can be used to evaluate the utilization of the topology and identify overloaded resources, for which subsequent analyses may not make sense.
To compute the latencies of messages, the model describing the Ethernet network is first transformed into an equivalent timing analysis model. This way, the Ethernet system can be modeled in "typical" network jargon (with switches (hops), ports, links and messages), but the analysis can operate on a typical timing model (with resource providers and consumers). The conceptualization of these models and their transformation has been performed in WP2.
The internal timing analysis model transformation is based on the following rule set:

- Each EthernetPort is transformed into a EthernetPortResource, i.e. a resource providing service of transferring data to the next hop. This resource is shared by all frames leaving a hop through the same port. Any contention between frames is accounted for at this resource.
- Each Switch is transformed into a EthernetSwitchResource, i.e. a resource providing service of internal switch traversal (from input to output port). With this resource, a non-constant (but static) transmission time inside the switch can be accounted for.
- An EthernetMacFrame is created at each internal EthernetPortResource the message traverses, i.e. a task consuming the service of transferring data to the next hop. The timing properties of the EthernetMacFrame are derived as follows:
  - The transmission time is calculated based on the frame size and port transmission speed.
  - The priority is the EthernetMessage priority.
  - The activation is the EthernetMessage activation.
- An EthernetSwitchFrame is created at each internal EthernetSwitchResource the message traverses, i.e. a task consuming the service of internal switch traversal.
  - The transmission time of the EthernetSwitchFrame is the switch delay.
- An EthernetInternalTrigger is created to link the connected EthernetFrames. This models event-driven activation of Ethernet frames (i.e. the transmission of a frame on one hop is activated by the completion of the transmission of the frame on the previous hop).
- An EthernetPath is created and the EthernetFrames of one transmission are mapped to it. Thus, the latency of this path is equivalent to the latency of the corresponding message.

Figure 4.13 shows an example of the internal timing analysis model for a simple network with two Electronic Control Units (ECUs) , a single switch and one message. During model transformation, the timing analysis model will "stay connected" to non-Ethernet model elements, such as tasks on ECUs that trigger the transmission of an Ethernet message. This way, a complete system analysis can be performed.
The model transformation may also require a consideration of message fragmentation, which happens if the size of a message is larger than the Maximium Transfer Unit (MTU) of the sender. In this case, a single message is divided into multiple Ethernet frames. Note that the timing analysis model only knows Ethernet frames, so a transformation of the activation event model is required, because one activation of the Ethernet message results in multiple Ethernet frames being activated.
After model transformation, the analysis is performed based on the approach presented in [**?**]: The routes (paths through the network), as well as all Ethernet frames and needed triggers, are generated by the internal model generator and will be deleted after the analysis run through. The routes of each Ethernet message are separately analyzed hop by hop. The worst-case latency analysis consists of 4 steps (without generation and cleaning of the timing analysis model):

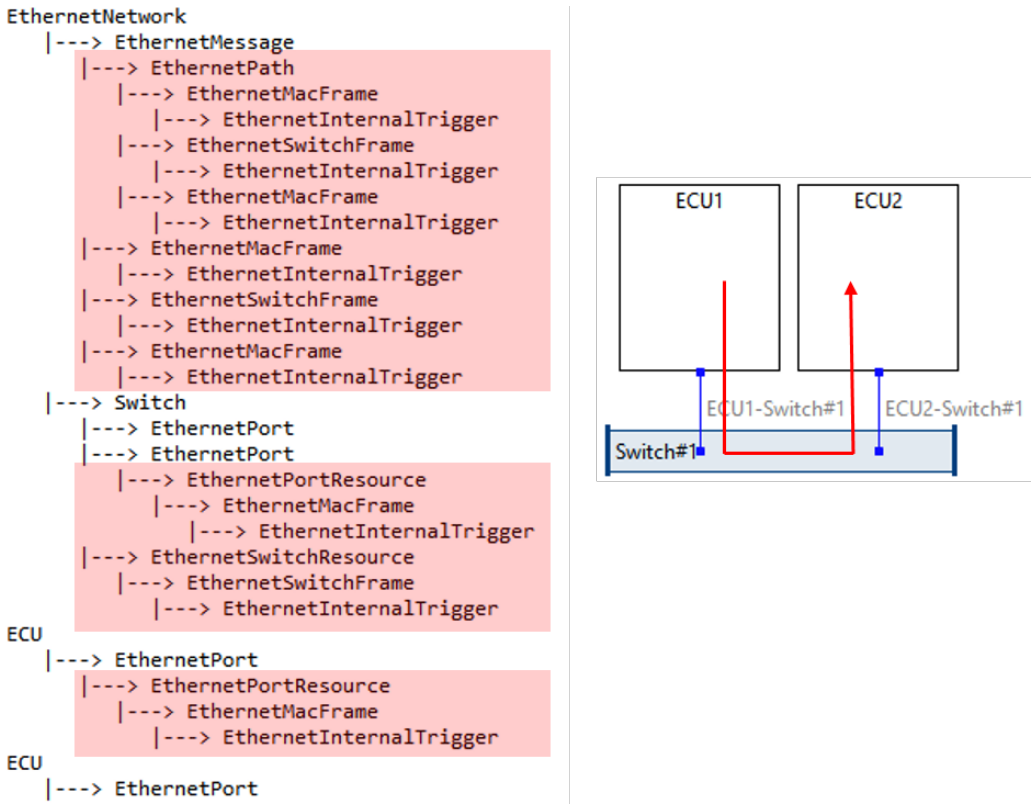


Figure 4.13: Example of a timing analysis model for a simple network with a single message.

1. EthernetMacFrames are analyzed message per message. A response time for the traversal of the output port is calculated for each of them.
2. The same is done for EthernetSwitchFrames. They are analyzed for each hop, producing a response time for internal switch traversal.
3. EthernetPaths are analyzed. For this, all response times of all hops of a route (computed in steps 1 and 2) are summed.
4. At last the response times for EthernetPaths are summarized for each Ethernet message. For this, the maximum response time of all routes (possible at multi- or broadcasted messages) is used as the worst case response time of the analyzed EthernetMessage.

The analysis produces various results, which are presented in the SymTA/S GUI as tabular results and/or charts.
Specifically, the results are (see Figure 4.14):

- Load of each port (item 4) and data rate for each message (item 2)
- Latency of each message (item 3)
- Path Gantt diagrams across Ethernet messages (item 1)
- Buffer requirements at each switch (work in progress)

4.5.2 Scheduling and Resource Management Algorithms

With respect to the topic of scheduling and resource management, there are several research topics that are of interest for mixed-critical Cyber-Physical System (CPS). Among those, we plan to investigate two topics:

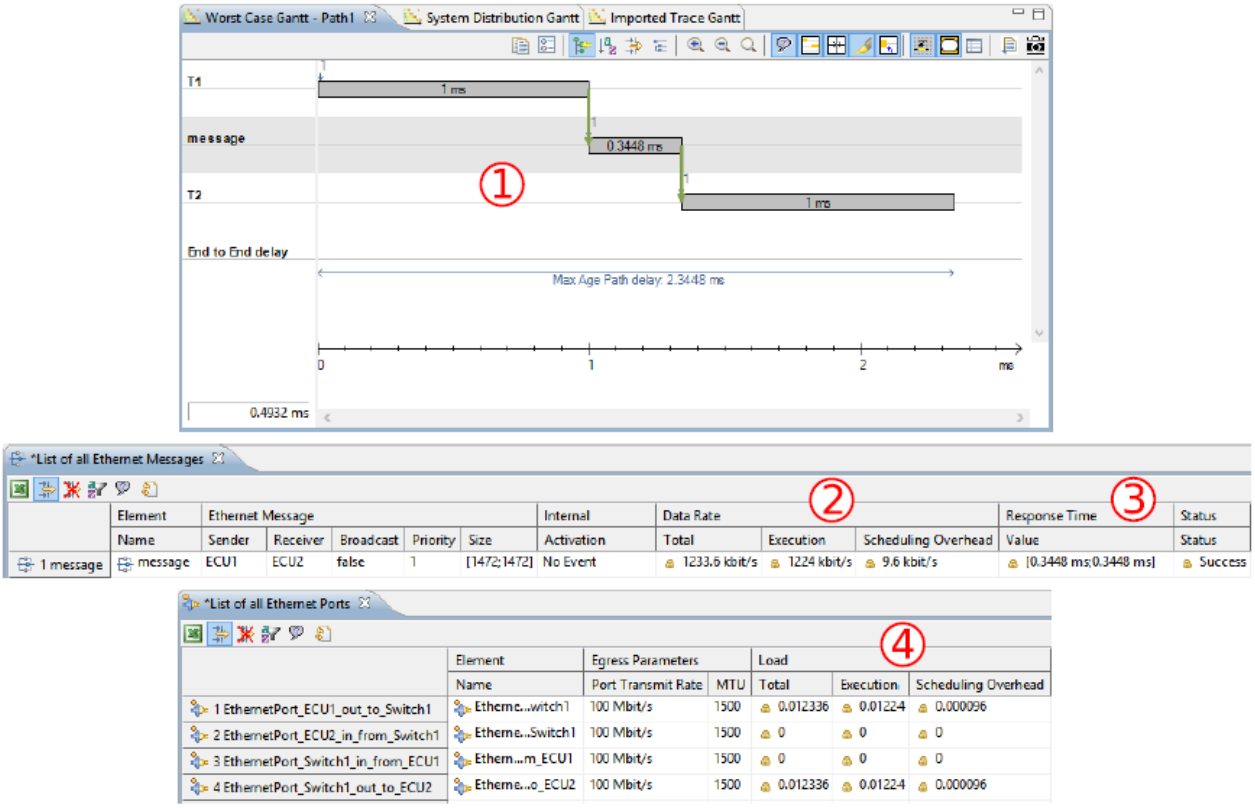


Figure 4.14: Results for an example system

- Scheduling and resource management algorithms for providing timing isolation and guaranteed processor bandwidth in multi-core systems and/or when end-to-end computations span across the boundaries of several resources (multi-core nodes and networks).
- Scheduling techniques for time-critical automotive systems, especially with respect to systems that need to perform with controlled degradation in (temporary) overload conditions.

With respect to the first topic, the traditional approach consists in guaranteeing timing isolation by strict time separation and time-triggered scheduling or scheduling in which each thread or partition is assigned a well-defined time slice. However, in single core scheduling, there are other options that have been demonstrated capable of providing time isolation that do not require the strict enforcement of time-slot scheduling (and ensuing possible inefficiency for unused time slots). These techniques are based on the concept of controlling the execution of threads by means of a (time) server that assigns a time budget to each thread. When the threads execute according to the assumptions on activation rates and worst-case execution time, the server allocates them the planned time budget. If a thread attempts at executing longer than assumed, or more often, then the server time depletes and the thread is suspended until the server planned replenishment time.
An example of such technique is the CBS server [**?**], that allows the server time to be scheduled by fixed priority or Earliest Deadline (EDF). The CBS server has been extended to deal with resources [**?**] and finally implemented as part of the Linux kernel [**?**]. The extension fo these mechanisms to multi-core systems has been investigated in the Actors project [**?**] using the concept of hierarchical scheduling and an analysis methodology is presented in [**?**]. The discussion of techniques for resource management in multi-cores is also in [**?**]. Finally, for the relevant case (for multi-core systems) in which the memory is a significant shared resource, the Memguard algorithm [**?**] has been proposed to guarantee performance isolation for real-time tasks, see Section 4.2.2.1.
With respect to the second topic, there are three other possible specific research targets of interest.

The first consists of the extension of a set of task models and analysis techniques that have been developed in recent years for applications such as engine control. In this case, a significant number of threads are activated when the engine crankshaft reaches a given position. The activation is not periodic, but with variable rate. This means that a given processor utilization at low speeds could become higher and lead to possible overloads unless the behavior of the task is changed at high revolution rates. These threads are called Adaptive Variable Rate and the schedulability analysis for them has been studied in several papers [**?**] [**?**] [**?**]. The basic scheduling model assumed a hard-type behavior (deadlines should never be missed). However, in many engine control applications selected deadline misses can be tolerated and a true performance analysis as well as studies for the scheduling behavior of these tasks in overload conditions are needed.
The second target consists of the study of analysis techniques for another model to control overload, also known as (m-k) scheduling. In this case, the scheduling algorithm and the analysis methodology are required to guarantee that at least m instances of the thread are executed within their deadlines in any sequence of consecutive k instances.
Finally, we plan to review the model of real-time scheduling and schedulability analysis for mixed-criticality systems. The conventional model studied by the research community assumes that threads have different criticality levels (to simplify, sometimes only two levels are assumed). High criticality threads have two execution time specifications, one for high-critical certifications, and another for low-criticality behaviors. The high criticality execution time should be guaranteed for all the highly critical tasks, at the possible expense of low criticality computations. However, this models has been recently challenged [**?**] [**?**], and new techniques should be developed to provide analysis techniques that are best suited to the industrial needs for highly critical systems.

4.6 Design for the Joint Consideration of Data and Timing Integrity

The need to meet integrity constraints of different nature in the same system reveals cross concerns that are better addressed by an integrated design process that can aim at the optimization of the software architectures with respect to safety and security constraints.
Examples of research papers in this direction are appearing in several contexts. With respect to the automotive domain, there are established standards for scheduling and communication, as part of AUTOSAR. Also, AUTOSAR has standardized the security mechanisms for adding MAC codes to messages. When messages have joint time and security constraints, the addition of a MAC is going to affect the utilization of network bandwidth, and creates a cross concern with respect to the network delays.
Several research proposals discuss the options for the implementation of authentication protocols over CAN buses, and a performance analysis and comparison can be found in [**?**]. In [**?**] the problem of how to optimize the design of a set of CAN messages with time and security constraints is discussed as an extension of previous work, such as [**?**], in which the problem of assigning identifiers to CAN messages with timing constraints is solved using a Mixed Integer Linear Programming (MILP) formulation, that jointly optimizes also the task allocation to ECU nodes, signal packing into messages and the priority assignments of tasks upon the ECUs.
When considering security constraints, the solution in [**?**] and its extension in [**?**] assume that ECUs are classified in receiving groups with an assigned security level, so that multiple receiving ECUs are allowed to share the same MAC contained in a message, that is, the ECUs share the same key. According to the analysis in [**?**] such group keying is also the best practice for ensuring a high security level in a CAN bus based system. ECUs in the same group share the same trust level, which should be specified by the manufacturer.
With respect to the security constraints, [**?**] assumes that the trust level and the reserved maximum MAC lengths for each receiving ECU group are given as parameters in the design problem formulation, and the final objective is to minimize the security risks (leveraging MAC truncation when required by the timing constraints). Below we list examples of security constraints that can affect the

task allocation and signal packing.

- A task can be allocated onto an ECU only if the ECU is in a receiving group in which the associated trust level is no lower than the value required by the task for receiving a secured message.
- A task can be allocated onto an ECU only if the reserved MAC length of the corresponding receiving group is not smaller than the one required by the task for receiving a secured message.
- Multiple MACs can be packed into the same message only if the data payload and required bits for MAC protection do not exceed the limit of a CAN bus frame.

Still, in [**?**] the security constraint is put on each signal/message transmission that needs security protection, and its objective function is to minimize the end-to-end latency of the functional path. This may not reflect the real security requirement in the embedded system, where the security requirement is associated with the functional path (e.g. from the sensor to the actuator) and an objective function to minimize the security risk along with the functional path becomes more meaningful. As a result, [**?**] extends the formulation in [**?**] to take such reasoning into account. To ease the complexity of the resulting problem definition, [**?**] simplifies the MILP formulation in [**?**]. As an example, a security constraint can be in the form such that the MAC lengths of all signals in a path are long enough or the security risk of a functional path, which passes by several different receiving groups with different trust levels, should not exceed a given level.
The MILP approach provides an optimal solution, at the cost of a possible high computational complexity. Heuristics have also been studied in [**?**] and [**?**] for the security-aware mapping of functional paths: signal packing, task/message priority assignment and ECU/receiving group assignment.
When more than one authentication protocol is available in the CAN system, the above discussed security-aware mapping of functional path can also be applied for selecting the best one, as discussed in [**?**]. For security mechanism selection, at first each corresponding security-aware mapping problem can be solved, then the one with best performance will be selected. On the other side, if different security mechanisms can be abstracted into a common set of variables and parameters, then it is possible to efficiently choose the best one according to one single security-aware mapping problem.
In SAFURE we plan to extend this line of research by explicitly considering the AUTOSAR recommendations and requirements and by studying the impact of different security requirements applied at the functional/application level, as defined in WP2.

Chapter 5 Conclusion

This document overviews the different integrity aspects considered in SAFURE and their respective integrity algorithms. This document only presents interim analysis of integrity algorithms; where final analysis will be covered in D3.2. We consider three different aspects of integrity:

5.1 Temperature Integrity

In SAFURE we consider both threats to system safety and threats to system security posed by temperature. For the safety aspect, we consider the impact of scheduling tasks with multiple criticalities (different safety requirements) on a multicore platform. In this research thread, we will provide a method to analyze the peak temperature of mixed-criticality systems in different scheduling modes, with temperature dependencies across scheduling modes addressed. Our analysis will show how common mixed-criticality mechanisms could affect the system peak temperature.
For the security aspect, we have already studied the characteristics of thermal covert channel. On a modern mobile platform, we have shown that data rates of up to 50 bits per second are possible with an error probability of 0.1 percent. Going forward, we will analyze the covert channel in the telecommunication use case platform (Dragon board 810).

5.2 Data Integrity

In SAFURE we focus on ensuring data integrity by cryptographic mechanisms. While checksums are sufficient to detect erroneously transmitted data or defective memory areas, they can easily be adapted by an adversary who has manipulated the data. We are going to analyze the suitability of algorithms from different areas of cryptography, including symmetric and asymmetric approaches as well as algorithms based on finite field arithmetic and Elliptic Curve Cryptography, for mixed-critical systems. We want to examine to what extend the different cryptographic systems meet real-time requirements of safety-critical systems and how the benefits of a multicore platform can be utilized. Demo implementations of certain algorithms will be provided on the Dragon Board 810, which has been chosen as the demonstrator platform for the telecommunication use case.

5.3 Timing and Resource Sharing Integrity

SAFURE focuses on timing integrity on multiple areas: On multi-core architectures, interference between applications due to access of hardware shared resources is considered. For Ethernet networks, interference of different traffic streams in the network is considered. For both areas, state-of-the-art has been presented and analyzed. First approaches on vulnerability detection have been proposed and explained in this document. With these, an assessment is possible whether a given configuration or scenario results in illegal timing interference and thus a violation of timing integrity.
A future direction, which we plan to address in the Deliverable D3.3, is to build and/or analyze vulnerability protection mechanisms, which allow not only detection but also prevention of timing integrity violations.