4th International Workshop on Security and Dependability of Critical Embedded Real-Time Systems

CERTS 2019, July 9, 2019, Stuttgart, Germany

Edited by

Mikael Asplund
Michael Paulitsch

OASICS

Editors

Mikael Asplund
Linköping University, Sweden
mikael.asplund@liu.se

Michael Paulitsch
Intel, Germany
michael.paulitsch@intel.com

ACM Classification 2012
Security and privacy → Embedded systems security; Computer systems organization → Embedded and cyber-physical systems; Computer systems organization → Dependable and fault-tolerant systems and networks

ISBN 978-3-95977-119-1

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-119-1.

Publication date
July, 2019

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/OASIcs.CERTS.2019.0

OASIcs – OpenAccess Series in Informatics

OASIcs aims at a suitable publication venue to publish peer-reviewed collections of papers emerging from a scientific event. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

https://www.dagstuhl.de/oasics

Contents

Preface

This is the fourth iteration of the workshop on security and dependability of critical embedded real-time systems (CERTS) and it is co-located with the 31st Conference on Real-Time Systems (ECRTS'19) held at the premises of Bosch in Stuttgart, Germany on July 9-12, 2019. The technical program of the workshop includes multiple peer reviewed papers and a keynote from Dr. Borislav Nikolic from TU Braunschweig on the topic of Safety and Security aspects of TSN.

The aim of this workshop is to bring researchers and practitioners from a variety of domains, viz., real-time and embedded systems, security, dependability and cyber-physical systems to name just a few. The idea is to foster a community that looks at all of these topics and develops techniques, algorithms, policies and frameworks to improve the security and dependability of critical systems. We hope that the papers and the keynote will help foster such discussions and collaborations.

CERTS 2019 owes its success to a variety of people. We would like to thank the steering committee that consists of: Paulo Esteves-Verissimo, Marcus Völp, Antonio Casimiro and Rodolfo Pellizzoni. We would also like to thank the technical program committee members for taking the time to review and provide feedback for the papers. In addition, we also wish to thank the organizers of ECRTS 2019, in particular: Steve Goddard and Martina Maggio in the Organization Committee, Sophie Quinton (Program Chair and very helpful with web issues) and Gerhard Fohler (Real-Time Technical Committee Chair). Finally, we would like to thank the authors and participants of the workshop without whom this event would not be successful.

We hope that you will enjoy the CERTS 2019 program and that it will foster many new research directions and collaborations.

Michael Paulitsch
Intel Labs

Mikael Asplund
Linköping University

Workshop Organizers and Program Committee

Workshop Organizers:
Michael Paulitsch, Intel
Mikael Asplund, Linköpings Universitet

Technical Program Committee:
Antônio Augusto Fröhlich, Federal University of Santa Catarina, Brazil
Rakesh Bobba, Oregon State University, US
Christian Esposito, University of Naples Federico II, Italy
Marisol Garcia Valls, Universidad Carlos III de Madrid, Spain
Martin Gilje Jaatun, University of Stavanger, Norway
Karl Goeschka, Vienna University of Technology, Austria
Gert Jervan, Tallinn University of Technology, Estonia
Zbigniew Kalbarczyk, University of Illinois at Urbana-Champaign, US
Martina Maggio, Lund University, Sweden
Sibin Mohan, University of Illinois at Urbana-Champaign, US
Sasikumar Punnekkat, Maelardalen University, Sweden
Hans Reiser, Universität Passau, Germany
Soheil Samii, General Motors, US
Elena Troubitsyna, KTH, Sweden
Bryan Ward, MIT Lincoln Laboratory, US
Saman Zonouz, Rutgers University, US

Steering Committee:
Marcus Voelp, SnT – University of Luxembourg
Paulo Esteves-Verissimo, SnT – University of Luxembourg
Antonio Casimiro, University of Lisboa
Rodolfo Pellizzoni, University of Waterloo

Combined Security and Schedulability Analysis for MILS Real-Time Critical Architectures

Ill-ham Atchadam
University of Brest, Lab-STICC, CNRS UMR 6285, France
ill-ham.atchadam@univ-brest.fr

Frank Singhoff
University of Brest, Lab-STICC, CNRS UMR 6285, France
frank.singhoff@univ-brest.fr

Hai Nam Tran
University of Brest, Lab-STICC, CNRS UMR 6285, France
hai-nam.tran@univ-brest.fr

Noura Bouzid
University of Brest, Lab-STICC, CNRS UMR 6285, France
noura.bouzid@univ-brest.fr

Laurent Lemarchand
University of Brest, Lab-STICC, CNRS UMR 6285, France
laurent.lemarchand@univ-brest.fr

Abstract

Real-time critical systems have to comply with stringent timing constraints, otherwise, disastrous consequences can occur at runtime. A large effort has been made to propose models and tools to verify timing constraints by schedulability analysis at the early stages of system designs. Fewer efforts have been made on verifying the security properties in these systems despite the fact that sinister consequences can also happen if these properties are compromised. In this article, we investigate how to jointly verify security and timing constraints. We show how to model a security architecture (MILS) and how to verify both timing constraints and security properties. Schedulability is investigated by the mean of scheduling analysis methods implemented into the Cheddar scheduling analyzer. Experiments are conducted to show the impact that improving security has on the schedulability analysis.

2012 ACM Subject Classification Software and its engineering; Computer systems organization → Real-time system architecture

Keywords and phrases MILS (Multi Independent Levels of Security), RTCS (Real-Time Critical Systems), Security architecture and models, Scheduling analysis, Security analysis

Digital Object Identifier 10.4230/OASIcs.CERTS.2019.1

Funding This work and Cheddar[1] are supported by Brest Métropole, Ellidiss Technologies, CR de Bretagne, CD du Finistère and Campus France PESSOA programs 27380SA and 37932TF.

1 Introduction

Real-time critical systems (RTCS) are systems characterized by the temporal behaviors of their functions. The functions must imperatively respect deadlines specified by designers and serious disasters may occur if these deadlines cannot be met at runtime.

Although the respect of deadlines is an important issue in RTCS design, the security has also to be considered. The general purpose of securing a system is to prevent information disclosure (i.e. confidentiality) and preclude information's alteration (i.e. integrity) [22].

[1] `http://beru.univ-brest.fr/~singhoff/cheddar/`

Security architecture models such as MILS (Multiple Independent Levels of Security) have been designed to make RTCS compliant with security objectives [12]. However, it is a challenge to design RTCS architectures with both resources management to enforce deadlines and also security policies to ensure security properties.

Schedulability can be enforced by a proper assignment of the tasks on the processors [13]. However, task assignment may lead to modify the RTCS architecture because of the assignment of security levels to the components. Indeed, assignment of security levels may require extra components to avoid violation of the information flow security. Then security architecture decisions may impact schedulability of RTCS and some means have to be proposed to RTCS designers to predict the impact of security-driven architectural design choices on the task schedulability, i.e. on the met deadlines of RTCS.

The ultimate goal of our work is to provide methods and tools for the exploration of real-time critical and secure architectures. Starting from a predefined model with real-time and security specifications, we want to automate the search of the design space defined by the constraints embedded in those specifications. It allows us to propose a set of solutions that represent trade-offs between security and schedulability requirements. For the first steps, in this article, we propose to evaluate the conflict aspects between security and schedulability concerns.

In this article, we propose to integrate a security architecture (MILS) in the Cheddar schedulability analyzer, a tool allowing RTCS designers to predict if the deadlines will be met at runtime. We extend the modeling capabilities of Cheddar [21] to model MILS security aspects and we ensure RTCS security properties verification by implementing several security models (Bell-La Padula, Biba). The resulting tool allows RTCS designers to run combined security and schedulability analysis. In addition, in order to evaluate the impact that improving systems security has on RTCS schedulability, we build a complete case-study used to conduct some experiments. Finally, with these experiments, we show the drawback of enforcing security properties for the RTCS schedulability.

The rest of the article is organized as follows. Section 2 gives an overview of security architectures, security models and the MILS architecture. Section 3 presents the Cheddar scheduling analyzer and depicts the system model and assumptions considered in our work. In addition, we propose our extension of Cheddar to model the MILS architecture. Section 4 details our approach of implementing security models which help to verify MILS architecture compliance and to compute number of security violations. We present a case study and our experiments to validate our contributions aforementioned in Section 5. Finally, Section 6 discusses related work and Section 7 concludes the article.

2 Background

This section introduces the main aspects relevant to security architecture and models. An overview of a high-assurance security architecture named MILS is also presented.

2.1 Security architecture and models

Regarding architecture designing, the concept of layering consists on separating hardware and software features into modular levels (Hardware, Kernel and Device Drivers, Operating System and Applications) [6]. Systems may use classifications, such as United States government classification system [16], which is based on the degree of secrecy and level of sensitivity. Classification levels can be confidential, Top_secret and Secret. They can be applied to subjects or objects. Objects can be data classified at one level and subjects apply operations such as read, write or execute on objects.

A security model [22] describes the security strategy for a system with the purpose of ensuring security objectives (confidentiality, integrity). It is an implementation of some mathematical and analytical assumptions mapped to a system specification for resolving security issues. Examples of security models are Graham-Denning model [10], information flow control (IFC) models [22], State-Machine model [14], non-Interference model [9]. Bell-La Padula [3] and Biba [4] are concrete examples of IFC models.

A security architecture [6] uses an integrated view of the system to comply with security requirements. MLS (Multiple Levels of Security) and MILS (Multiple Independent Levels of Security) are examples of such security architecture [1].

2.2 MILS architecture

MILS is a high-assurance security architecture characterized by untrusted and trusted components, based on security models such as IFC [12]. It ensures confidentiality [22] which is the insurance of preventing the system from information disclosure and integrity [4] which is the protection of the system from unauthorized alteration. To meet those requirements, MILS is based on a set of properties named NEAT (Non-by-passable, Evaluatable, Always-invoked, Tamperproof) [2].

In order to ensure IFC, MILS adopts a classification level for subjects and objects which is Top_secret, Secret, Confidential and Unclassified for confidentiality; and High, Low, Medium for integrity. MILS is also based on the divide and conquer approach in order to reduce the effort for a system's security evaluation.

MILS introduces the following concepts [12]:

- Processes, which model the applications in a RTCS. For IFC purpose, they have to be labeled by a confidentiality level (e.g. Top_secret) and an integrity level (e.g. High). The same confidentiality and integrity levels are also applied to objects.
- Partitions, which are units of separations that host processes and/or data. They are characterized by a resource allocation in the space and time domains. Communications between processes in a partition and between partitions are subjected to IFC.
- Middleware Service Layer, which is a service responsible for maintaining the IFC. To enforce IFC, such component applies restrictions and permissions expressed by the designer according to communication between messages and processes and between processes themselves.
- Objects (*message*, *buffer*), communications.
- Application layer, which concerns all the processes. The components of this layer can be of 3 types[1]: Single Level of Security (SLS), Multiple Levels of Security (MLS), or Multiple Single Level of Security (MSLS).

3 MILS architecture modeling in Cheddar

In this section, we give explanations and details about Cheddar, a real-time scheduling analysis tool. We describe next how our models are represented and some assumptions adopted for this purpose. Last, we present the MILS modelling in Cheddar.

3.1 Cheddar scheduling analyzer

Cheddar [21] is a free real-time scheduling analysis tool. It provides a graphical editor and a library of schedulability analysis modules. The entry point of the tool is an architecture model expressed with Cheddar ADL. Cheddar ADL is a dedicated language designed to model software architecture of RTCS for scheduling analysis with Cheddar.

Cheddar ADL's basic entities can be grouped into 2 types: hardware and software components. The formers model the execution platform of the RTCS to be analyzed. Such components allow the designer to model processors, cache units, cores or any computing units, memory units and communication units. Software components model the software entities, i.e. applications composing a RTCS. Those components can be entities to model flows of control (Cheddar ADL *Task* entity), shared resources (entity *Resource* or *Buffer*) or task dependencies (Cheddar ADL entity *Dependency* or *Message*).

From a Cheddar ADL model, Cheddar provides two kinds of analysis tools: feasibility analysis tool, which assesses schedulability by analytical methods; and simulation analysis tool, which assesses schedulability by scheduling simulations.

3.2 System model and assumptions

We assume a RTCS consisting of tasks scheduled by a preemptive fixed priority scheduler. The system is compliant with the MILS architecture and is defined as follows:

- $\tau_1, \tau_2, ..., \tau_i, ..., \tau_n$ are the n tasks composing the system. Each task models a MILS process.
- T_i, C_i, D_i are respectively the period, the capacity (or the worst-case execution time) and the deadline of task τ_i. Any task τ_i makes its initial request at time 0, and then released periodically every T_i units of time. The task set is then synchronous. A task requires C_i units of computation time and must complete before D_i units of time.
- O_1, O_2, ..., O_i, ..., O_m are the m MILS objects in the system. Objects model messages exchanged between the tasks. Tasks can have read or write access to messages.
- IL_i and CL_i are respectively τ_i or O_i integrity and confidentiality levels.
- dep(τ_i, τ_j) describes a communication from task τ_i to task τ_j.
- dep(τ_i, O_j) describes a communication from task τ_i to object O_j, task τ_i having write access to the object O_j.
- dep(O_j, τ_i) describes a communication from object O_j to task τ_i, task τ_i having read access to the object O_j.

In order to model MILS architectures with Cheddar, we need to make some extensions to Cheddar ADL, even if several Cheddar's components can be already used to model few MILS's entities:

- MILS partitions can be modeled by Cheddar *Address_space* entities.
- MILS objects can be modeled by Cheddar shared resources (i.e. *Buffer* or *Message* entities).
- MILS processes can be represented by Cheddar $Tasks$.
- MILS communications have the same semantics than Cheddar ADL dependencies. MILS messages can be modeled by asynchronous communication Cheddar dependencies.
- MILS functions do not need to be represented in Cheddar as they can be modeled by groups of partitions (i.e. groups of Cheddar Address Spaces).
- Finally, applications (which can be composed of one or more processes) are modeled by Cheddar ADL *Task* entities.

To complete the modeling capabilities of Cheddar for MILS (Fig. 1) architectures, several Cheddar ADL entities have to be extended with new attributes modeling MILS data. As an example, we need to model *Buffers* and *Messages* confidentiality and integrity levels and also the right levels of tasks and partitions that are using them, i.e. if a partition or a task is allowed to handle a *Buffer* or a *Message* according to its authorization levels. We give bellow the list of properties we actually added to Cheddar ADL entities:

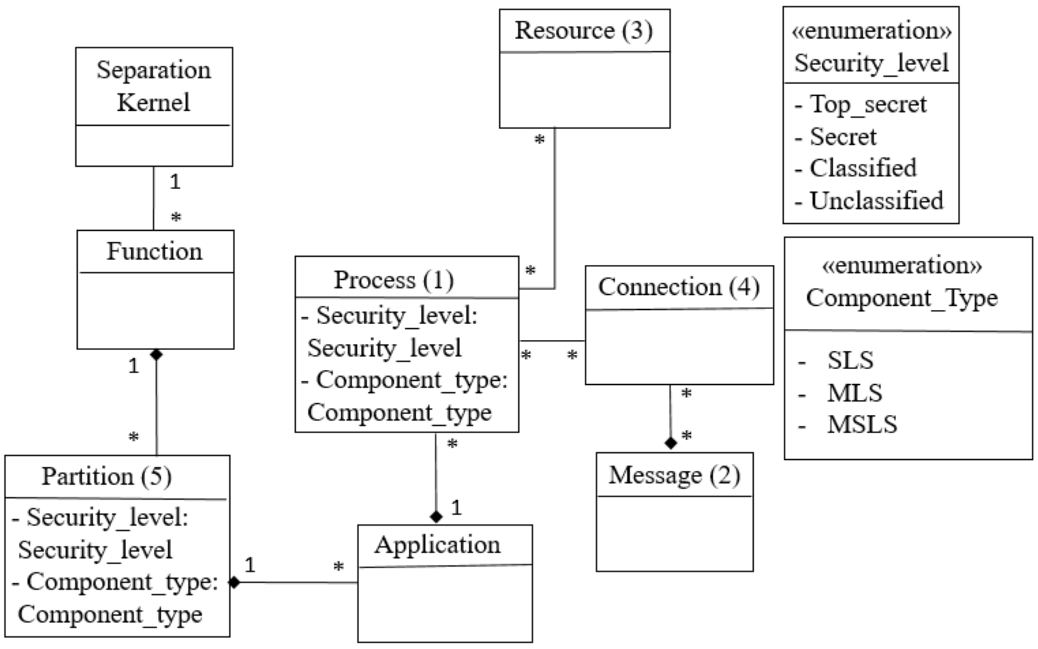
Separation Kernel
Function
Partition (5)
- Security_level: Security_level
- Component_type: Component_type
Process (1)
- Security_level: Security_level
- Component_type: Component_type
Resource (3)
Application
Connection (4)
Message (2)
«enumeration» Security_level
- Top_secret
- Secret
- Classified
- Unclassified
«enumeration» Component_Type
- SLS
- MLS
- MSLS
1
*

Figure 1 Modeling of MILS.

- An attribute named *Confidentiality_Level* (Top_secret, Secret, Classified, Unclassified) has to be defined for each Cheddar ADL address spaces, objects and tasks entities.
- An attribute named *Integrity_Level* (High, Medium, Low) also has to be defined for each Cheddar ADL address spaces, objects and tasks entities.
- *MILS_component_type* (SLS, MLS, MSLS) is an attribute to model MILS type of security level. Again, such attribute has to be defined in tasks and address spaces Cheddar ADL entities.
- Finally, *MILS_compliant_type* (Non_Compliant, Partition,...) specifies if a Cheddar's entity models a MILS's component or not. Such attribute is defined in any Cheddar ADL entity.

4 MILS security model implementation

In order to verify MILS architectures, we have implemented the two well-known security models Bell-La Padula [3] and Biba [4] into Cheddar. Both of them have been adapted and implemented in order to verify Cheddar ADL models. In the sequel, we describe their implementation as functions that take as entry a Cheddar model composed of cores, processors, address spaces, buffers, tasks, and dependencies.

4.1 Bell-La Padula in Cheddar

This security model was introduced to formalize the U.S. Department of Defense (DoD) multilevel security [16]. It is based on the *No read up, No write down* principle. No read up refers to the fact that a subject at a given security level cannot read data that is tagged with a higher security level. No write down means that a subject tagged with a given security level cannot write information to a lower security level.

We have implemented the Bell-La Padula algorithm in Cheddar. It checks if a Cheddar ADL model conforms to Bell-La Padula rules by returning the number of No read-up/No write-down rule violations. The algorithm is sketched below:

```
Sys: a Cheddar model composed of n tasks ($\tau_1, \tau_2, ..., \tau_i, ..., \tau_j, ..., \tau_n$)
num: number of rules violations
For each dep in Sys loop
        If ((dep = dep($O_j, \tau_i$)) OR (dep = dep($\tau_i, O_j$)))
        AND ($CL_i$ < $CL_j$) Then
                num:= num+1
        Elsif (dep = dep($\tau_i, O_j$)) AND ($CL_j$ < $CL_i$) Then
                num:= num+1
        Elsif (dep = dep($\tau_i, \tau_j$)) AND ($CL_j$ < $CL_i$) Then
                num:= num+1
        End if
End loop
Return (num)
```

4.2 Biba in Cheddar

This security model was developed to ensure data integrity [4]. It is based on the *No read down, no write up* principle. "No read down" means that a subject cannot read data from a lower integrity level and "No write up" means that a subject cannot write data to an object tagged with a higher integrity level. We have implemented Biba rule checking into Cheddar. The algorithm is sketched below as a function that checks if a Cheddar model conforms to Biba rules by returning how many times the Read down and Write up rules are missed.

```
Sys: a Cheddar model composed of n tasks ($\tau_1, \tau_2, ..., \tau_i, ..., \tau_j, ..., \tau_n$)
num: number of rule violations
For each dep in Sys loop
        If (dep = dep($O_j, \tau_i$)) AND ($IL_j$ < $IL_i$) Then
                num:= num+1
        Elsif(dep = dep($\tau_i, O_j$)) AND ($IL_i$ < $IL_j$) Then
                num:= num+1
        Elsif (dep = dep($\tau_i, \tau_j$)) AND ($IL_i$ < $IL_j$) Then
                num:= num+1
        End if
End loop
Return (num)
```

4.3 MILS securing process

In order to design MILS architectures that meet the different security rules previously presented, we propose to add the following components to the architecture model.

When a communication between two tasks of different security levels doesn't meet a security rule, we propose to add an encrypter or a decrypter components between the two tasks. So the tasks can still communicate but the data sent are encrypted and the designer can decide to allow communications by using a decrypter on the other side of the communication link. In MILS, such an encrypter (resp. a decrypter) is called a downgrader (resp. an upgrader).

In our implementation, downgraders and upgraders are extra Cheddar ADL tasks. Each of them has a period, a capacity, a deadline, a confidentiality and an integrity level. Assuming a Cheddar model composed of a set of tasks and their related dependencies, we run through these dependencies and check if they meet Bell-La Padula or Biba rules. If not, we add a downgrader or upgrader depending if it is Bell-La Padula or a Biba violation.

For such purpose, we make an arbitrary assumption about the downgrader (resp. upgrader) parameters: downgraders (resp. upgraders) inherit from receiver task period and deadline.

Furthermore and for the sequel experiments, we have chosen for encryption the AES/GCM (2K tables) algorithm (cycle per Byte: 16.1; cycles to set up key and IV(Initialization Vector): 3227) [7]. By supposing that our data size is 64 Bytes and that the frequency is 125 MHz, then the encrypter's execution time is 1660 us.

5 Experiments & Evaluation

The objective of our experiments is to evaluate the integration of the security architecture MILS into Cheddar and more precisely the two security models Bell-La Padula and Biba implemented into Cheddar. We also evaluate the impact of improving RTCS security by applying MILS can have on the RTCS schedulability.

We assume a single processor execution platform, on which all the tasks are placed in the same partition. The task set is synchronous and scheduled by preemptive fixed priority scheduler. Each task has two possibilities of *Confidentiality_Level* (Top_secret or Secret) and *Integrity_Level* (High or Medium).

The experimentation process is built by the three following parts: non secure models generation, secure models generation and models evaluation.

In the remainder of this section, we present in 5.1 a case study made for our experiments purpose. Sections 5.2 and 5.3 refer to the models generation part of the experiments. Finally section 5.4 gives the evaluation of the generated models and also the results of the experiments.

5.1 Case study

To conduct the experiments, we have created the study case 2 inspired by a drone system. It is made of two applications composed of dependent tasks: (1) a digital signal processing application called Constant False Alarm Rate detection (CFAR) [19]. It is a set of low critical tasks designed to detect target based on the variation of background noise. (2) A flight control system called ROSACE (Research Open-Source Avionics and Control Engineering)[17]. It is a longitudinal and multi-periodic flight controller designed as a benchmark to respond to stability, real-time and performance issues. ROSACE is composed of high critical tasks.

The set of tasks and their parameters of our case study are described in Table 1. This table does not give confidentiality and integrity levels of tasks since they are decided later in our experiments.

5.2 Non secure models generation

This part consists of generating as many models as possible without considering if the generated models are secure or not. Our starting point is to describe a system. We used the case-study described in Section 5.1 with its default parameters of Table 1.

Confidentiality level of each task can be selected between two levels (Top_secret and Secret). The same holds for the integrity level (High and Medium). So our generator, for Bell-La Padula (resp. Biba) evaluation, takes the case-study model with an integrity (resp.

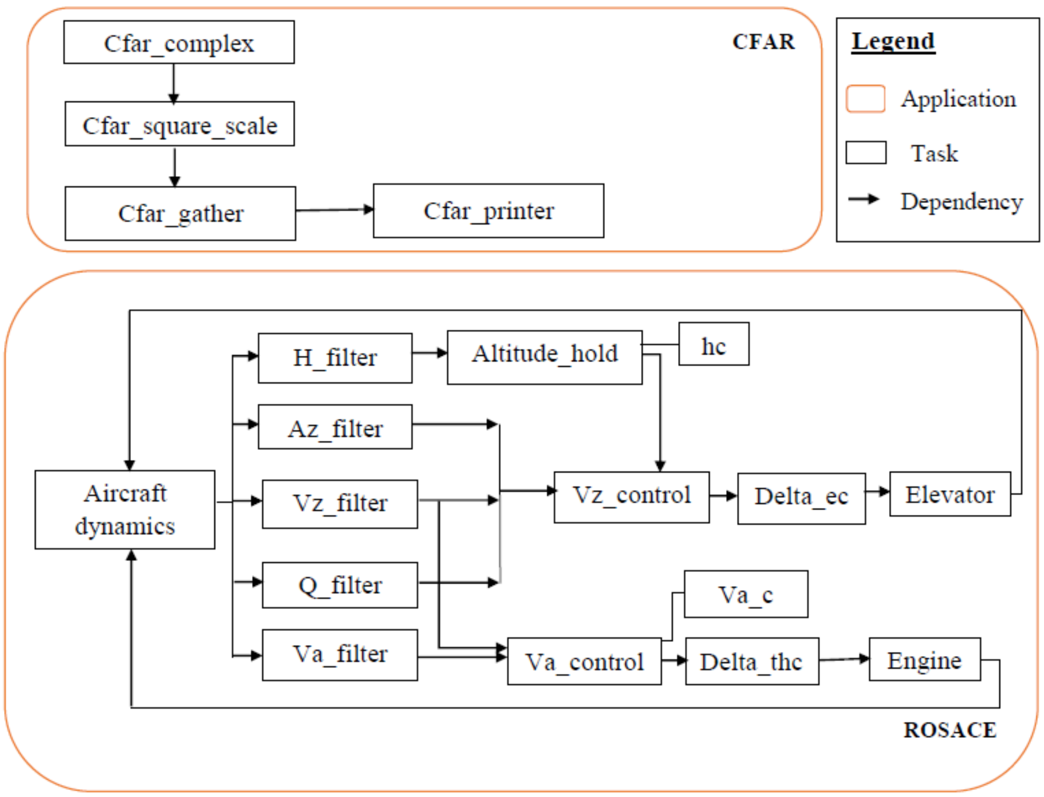

Figure 2 Case study model.

Table 1 Case study parameters.

Tasks	**Period/deadline [us]**	**Capacity[us]**
Cfar_complex	10000	90
Cfar_square_scale	10000	50
Cfar_gather	10000	340
Cfar_printer	10000	30
Aircraft dynamics	5000	200
Va_c, h_c	20000	500
H_filter, Az_filter, Va_filter, q_filter, Va_filter	10000	100
delta_e_c, delta_th_c	20000	500
Altitude_hold, Va_control, Vz_control	20000	100
Engine, Elevator	5000	100

confidentiality) level fixed for all the tasks and switches the confidentiality (resp. integrity) level of task. Each switching leads us to another different model. With 19 tasks in our case-study, we produce 1048576 ($2*2^{19}$) models for both Bell-La Padula and Biba evaluation.

5.3 Secure models generation

It consists of applying MILS securing process 4.3 on each non secure model generated in the previous step. To make all the non secure models secure, whenever a dependency doesn't respect Bell-La Padula or Biba's rules, we add a downgrader or an upgrader as additional task. For a given model, the number of downgraders or upgraders added is equal to the number of security rules violations.

As described in Section 4.3, we choose for encryption an AES/GCM (2K tables) algorithm with an execution time of 1660 us. Furthermore, we make an arbitrary assumption about the downgrader (resp. upgrader) attributes: downgraders (resp. upgraders) inherit from receiver task period and deadline.

5.4 Models evaluation

This section describes the evaluation of the schedulability and the security of all the generated models (non secure and secure). For each model, the security is quantified by the number of security violations while we decide to quantify the schedulability by the number of tasks missed deadlines.

It is important to notice that the number of security violations for secured models is null, while the number of missed deadlines will be null for the non secure models if the starting model is schedulable. Missed deadlines occur when adding security-related tasks (i.e. downgraders, upgraders).

As it can be noticed throughout this article, we have settled the security aspect into two parts: confidentiality and integrity. The figure (Fig. 3) below shows the relationship between scheduling (missed deadlines) and confidentiality (number of Bell-La Padula's rules violations).

In Fig. 3, each point corresponds to the number of confidentiality rules violations for a testcase (combination of different security levels) and the number of deadlines missed when we apply security rules to make the testcase secure by adding some downgraders.

The same work has been done for the integrity aspect. Figure 4) shows the relationship between schedulability (missed deadlines) and integrity (number of Biba's rules violations).

We observe that two different RTCS models with the same number of rules violations may not lead to the same number of missed deadlines. It is explained by the fact that downgraders/upgraders added to solve security problems may not have the same period and deadline in both RTCS models. Indeed, a downgrader's period and deadline depend on the period and the deadline of the sink task of the non-secure dependency.

From our experiments, with the security metrics, we conclude that the higher number of violations (confidentiality or integrity) is, the more we have tasks missing their deadlines when adding downgraders/upgraders through the MILS securing process. Fig. 3 and 4 show an quantification of such impact.

Finally, we can observe in the figures that even if the Bell-La Padula and Biba results of our experiments look similar, for a RTCS that violates both Bell-La Padula and Biba rules, resolving the Bell-La Padula's problem does not solve automatically the Biba's one and conversely.

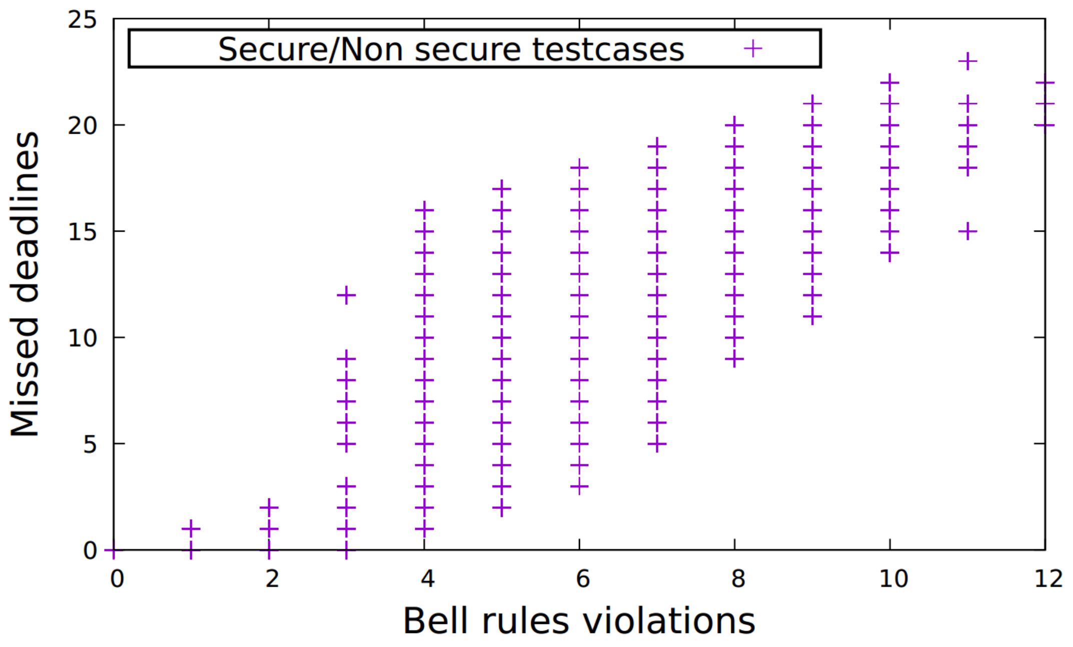

Figure 3 Schedulability and confidentiality evaluation.

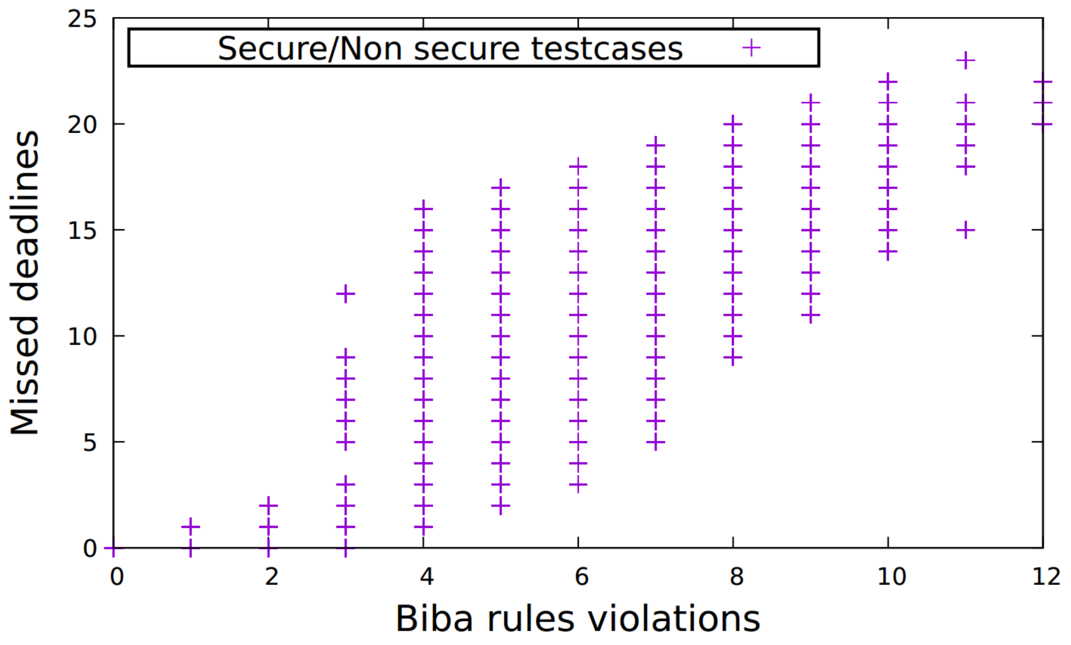

Figure 4 Schedulability and integrity evaluation.

6 Related work

In [15], the authors worked on designing a security gateway in avionic domain based on Integrated Modular Avionics (IMA) architecture, characterized by the separation of system resources into partitions. In order to control information flow between partitions for security issues, they applied MILS principles based on the real-time OS named PikeOS.

In [8], the authors modeled MILS using AADL (Architecture Analysis and Design Language) [20] and proposed an automated implementation process based on code generation. Applications are run on a MILS operating system called POK. In [5], the authors presented a guide on securing a system design using MILS architecture. They focused their work on distributed systems and systems of systems.

The article [11] discusses how to build a secure system by using AADL. The authors proposed an approach to validate the confidentiality of systems by defining assumptions based on Bell-La Padulla's security protocol. They showed how security can have impact on data quality, resource consumption, availability, reliability, and real-time performances.

One important concern about MILS is task partitioning. [18] presents XtratuM, a hypervisor that helps to build partitioned systems while meeting safety critical real-time requirements. Scheduling analysis is performed using Xcronte. The authors also measure the cost in terms of performance and memory footprint of using XtratuM compared to other partition development environments.

Our approach is different of the above ones because we integrate MILS concepts in scheduling analysis as none of them have experimented joined schedulabity and MILS security analysis . As far as we know, it is the first time one extends a scheduling analyzer to support a secure architecture. We implemented into Cheddar several security models in order to jointly enforce MILS properties and schedulability.

7 Conclusion

In this article, we investigated security properties in RTCS through a high assurance security architecture MILS. Our work overlays real-time scheduling analysis, and architecture design and security.

We have proposed an extension of Cheddar to model MILS real-time critical architecture and to perform both schedulability analysis and security analysis. We have integrated MILS concepts including the two security models (Bell-La Padula and Biba) with Cheddar ADL.

We have evaluated the impact of security on schedulability analysis. From the experiments, we observed that securing a system can affect negatively its scheduling by leading some tasks missing their deadlines. We also have shown that downgraders and upgraders periods and deadlines have impact on numbers of missed deadlines. Finally, we observed that confidentiality and integrity are independent, meaning that resolving one does not help to resolve the other.

As future work, we want to measure more precisely the cost of securing a RTCS and explore trade-offs to optimize both security and scheduling with different encryption algorithms for downgrader and upgrader components.

References

1 Jim Alves-Foss, Paul W Oman, Carol Taylor, and Scott Harrison. The MILS architecture for high-assurance embedded systems. *IJES*, 2(3/4):239–247, 2006.

2 R William Beckwith, W Mark Vanfleet, and Lee MacLaren. High assurance security/safety for deeply embedded, real-time systems. In *Proceedings of the Embedded Systems Conference*. Citeseer, 2004.

3 D Elliott Bell and Leonard J La Padula. Secure computer system: Unified exposition and multics interpretation. Technical report, MITRE CORP BEDFORD MA, 1976.

4 Kenneth J Biba. Integrity considerations for secure computer systems. Technical report, MITRE CORP BEDFORD MA, 1977.

5 Carolyn Boettcher, Rance DeLong, John Rushby, and Wilmar Sifre. The MILS component integration approach to secure information sharing. In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, pages 1–C. IEEE, 2008.

6 Eric Conrad, Seth Misenar, and Joshua Feldman. *Eleventh Hour CISSP*. Elsevier, 2013.

7 Wei Dai. Crypto++ 5.6.0 Benchmarks. URL: `https://www.cryptopp.com/benchmarks.html`.

8 Julien Delange, Laurent Pautet, and Fabrice Kordon. Design, Verification and Implementation of MILS systems. In *Proceedings of the 21th International Symposium on Rapid System Prototyping*, pages 1–8, 2010.

9 Lei Gong, Lu Tian, and Fulian Zhang. Application information flow non-interference transmission model. In *Proceedings of 2011 Int. Conf. on Electronic & Mechanical Engineering and Information Technology*, volume 5, pages 2306–2309. IEEE, 2011.

10 G Scott Graham and Peter J Denning. Protection: principles and practice. In *Proceedings of Spring Joint Computer conference*, pages 417–429. ACM, 1972.

11 Jörgen Hansson, Peter H Feiler, and John Morley. Building secure systems using model-based engineering and architectural models. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2008.

12 OH Holger Blasum and S Tverdyshev. Euro-mils: Secure european virtualisation for trustworthy applications in critical domains–formal methods used.

13 Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

14 Francois Mouton, Alastair Nottingham, Louise Leenen, and HS Venter. Underlying finite state machine for the social engineering attack detection model. In *2017 Information Security for South Africa (ISSA)*, pages 98–105. IEEE, 2017.

15 Kevin Müller, Michael Paulitsch, Sergey Tverdyshev, and Holger Blasum. MILS-related information flow control in the avionic domain: A view on security-enhancing software architectures. In *IEEE/IFIP Int. Conf. on Dependable Systems and Networks Workshops (DSN 2012)*. IEEE, 2012.

16 Barack Obama. Executive Order 13526: Classified National Security Information. In *United States. Office of the Federal Register*. United States. Office of the Federal Register, 2009.

17 Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. The ROSACE case study: From simulink specification to multi/many-core execution. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 309–318. IEEE, 2014.

18 Ismael Ripoll, Miguel Masmano, Vicent Brocal, Salvador Peiró, Patricia Balbastre, Alfons Crespo, Paul Arberet, and Jean-Jacques Metge. Configuration and Scheduling tools for TSP systems based on XtratuM. *Data Systems In Aerospace (DASIA 2010)*, 2010.

19 Benjamin Rouxel and Isabelle Puaut. STR2RTS: Refactored StreamIT benchmarks into statically analyzable parallel benchmarks for WCET estimation & real-time scheduling. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

20 SAE. Architecture Analysis & Design Laguage v2.0 (AS5506), September 2008.

21 Frank Singhoff, Jérôme Legrand, Laurent Nana, and Lionel Marcé. Cheddar: a flexible real time scheduling framework. In *ACM SIGAda Ada Letters*, volume 24. ACM, 2004.

22 Ming-Xin Yang, Li-Na Yuan, and Zhi-Xia Yang. A discuss of computer security strategy models. In *2010 Int. Conf. on Machine Learning and Cybernetics*, volume 2, pages 839–842. IEEE, 2010.

System Calls Instrumentation for Intrusion Detection in Embedded Mixed-Criticality Systems

Marine Kadar
SYSGO GmbH, Klein-Winternheim, Germany
marine.kadar@sysgo.com

Sergey Tverdyshev
SYSGO GmbH, Klein-Winternheim, Germany
sergey.tverdyshev@sysgo.com

Gerhard Fohler
Technische Universität Kaiserslautern, Germany
fohler@eit.uni-kl.de

Abstract

System call relative information such as occurrences, type, parameters, and return values are well established metrics to reveal intrusions in a system software. Many Host Intrusion Detection Systems (HIDS) from research and industry analyze these data for continuous system monitoring at runtime. Despite a significant false alarm rate, this type of defense offers high detection precision for both known and zero-day attacks. Recent research focuses on HIDS deployment for desktop computers. Yet, the integration of such run-time monitoring solution in mixed-criticality embedded systems has not been discussed. Because of the cohabitation of potentially vulnerable non-critical software with critical software, securing mixed-criticality systems is a non trivial but essential issue. Thus, we propose a methodology to evaluate the impact of deploying system call instrumentation in such context. We analyze the impact in a concrete use-case with PikeOS real-time hypervisor.

2012 ACM Subject Classification Security and privacy → Embedded systems security; Security and privacy → Intrusion detection systems

Keywords and phrases Instrumentation, Mixed-criticality, Real-Time, System Calls, Host Intrusion Detection Systems

Digital Object Identifier 10.4230/OASIcs.CERTS.2019.2

Funding This work has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 764785, FORA—Fog Computing for Robotics and Industrial Automation.

Acknowledgements The authors would like to thank the anonymous reviewers of the paper at CERTS and internal reviewers at SYSGO for their valuable feedback.

1 Introduction

Mixed-criticality systems consolidate software applications of different criticality levels on a single hardware platform. For example in a car, an Android operating system which manages infotainment can run together on a single hardware, with an AUTOSAR adaptive runtime environment that controls lights management. In such system, security is a major and non-trivial issue at design- and run-time. On one hand, software components with a low criticality (e.g., Android) offer full management and handling of critical resources (e.g., network, console I/O, file-system access) to their non-critical application workloads. On the other hand, highly critical software components running in parallel depend on the availability and correct operations of the same resources or shared hardware.

The intrusion detection approach assesses that the system environment potentially contains flaws which could be exploited by an attacker. To identify suspicious activity, this technique continuously observes the system at runtime. Suspicious activity corresponds either to an anomaly, defined as a deviation from normal activity, or to an intrusion, which is identified using the signature of previously known intrusions. Intrusion detection presents a strong potential as it can contribute to detect zero-day attacks. Intrusion Detection Systems (IDS) are notably used for securing IT infrastructures (e.g. anti-virus software, network monitoring, host behavior analytics). They also represent an important segment of research. In Host Intrusion Detection Systems (HIDS), system call patterns appear to be a good metric to identify security threats. System call monitoring has indeed been investigated since late nineties, starting with [20]. Recent research uses these data mainly for machine-learning based HIDS.

To the best of our knowledge, existing machine-learning based HIDS from the industry and research focus on deployment for generic computers. The use of such defense in a mixed-criticality system has yet not been investigated. Protecting embedded mixed-criticality systems is however essential, as these systems are exposed to same threats than non-critical systems. Adapting HIDS in this context is very challenging. One one hand, because it is deployed at system level, HIDS can potentially compromise system execution; e.g. a real-time application missing its deadline. On the other hand, HIDS can degrade time performance of non critical workloads. Thus, we need to ensure that the add-on does not alter system execution and constraints.

To implement system call monitoring, we have to integrate system call tracing in the mixed-criticality OS kernel. The new feature should be fast and predictable; its impact on real-time system execution needs to be determined. We propose an evaluation of the feasibility of system call instrumentation for deployment in a mixed-criticality system. After introducing our approach to evaluate the impact, we apply it in a concrete example, using the real-time hypervisor PikeOS [8].

The remainder of the paper is organized as follows. In section 2, we summarize related work. Section 3 introduces the methodology. Section 4 describes the experiment, while section 5 presents the results. We discuss in section 6 the security impact of intrusion detection as well as further implementation of the detection, We also mention other interesting data to detect intrusions as well as their potential impact on system's real-time constraints. Finally, section 7 concludes this paper.

2 Related Work

2.1 Security by Design

One main approach to enforce security, as well as safety, in mixed-criticality systems is to enforce the whole system's design, by providing strong isolation for all workloads. The OS kernel must be as protected in term of security and safety, as the most critical workload. To address this need, Multiple Independent Levels of Security (MILS) [2] architecture is a concept based on separation kernel and partitioning to ensure data and program separation. Nevertheless, software isolation can not be enough to ensure system isolation. The misuse of shared hardware resources by an attacker can potentially lead to system exploit, such as side channels attacks. For example, Spectre [12] and Meltdown [15] attacks succeed in leaking the memory of complete programs, by leveraging specific CPU vulnerability. As explained in [10], isolation of hardware resources is a non-trivial problem; because of system complexity, we generally can not assess 100% isolation. Hence, we need further protection to ensure system security.

2.2 Control Flow Integrity

Control Flow Integrity is a type of intrusion detection solution based on continuous monitoring. It observes programs executed in the system at run-time to detect any deviation from a defined normal execution policy. The policy can be defined for each individual program at compilation time, using specific tools such as LLVM compiler [18]. It generally provides a list of allowed branch transfers in the program address space.

TrackOS [17] is an example of real-time OS (RTOS), which integrates a CFI monitor. The RTOS hosts different workloads called tasks. One main task with higher execution privilege, the CFI Monitor, monitors all the other ones at run-time. Motivated by certification reasons, the monitored programs remain unchanged. A graph of execution is generated with static analysis from each untrusted program executable. Time credit is allocated by the user for each task, including the CFI Monitor. During its execution time slot, the monitor checks the stack of monitored workloads using the statically defined policy. Hence, the security impact on execution time for monitored workloads is controlled by the user.

Because of a higher frequency of control flow transfers over system call instructions, The impact of CFI on performance represents a major limitation to its deployment. For this reason, we do not consider CFI approach for our research.

2.3 System Call Tracing for Security

Host Intrusion Detection System (HIDS) is a well developed security solution, which relies on system call collection at runtime. It has been well investigated for two decades, since [20] highlighted a statistical correlation of intrusion patterns with certain system calls sequences. In literature, many papers focus on HIDS performance in terms of detection precision and false positive rates. It is an efficient way to detect intrusions; detection precision is generally above 90%, despite a high false positive rate, which can easily reach 15% in recent research. For example [9] compares several analysis models with average 90% detection precision and 15% false positives rates. [13] develops and HIDS based on Markov chains with 90% detection precision and 20% false positives rate. False postives can be reduced by modifying the model's parameters. Nevertheless, this usually induces a loss of precision: [5] compares six models on the same dataset, five of which show a lower false positive rate below 5%, while the detection rate varies between 40% and 100%.

Unfortunately, very few publications focusing on HIDS describe or analyze the impact on time execution for integrating such software at system run-time. [16] presents a short evaluation of the impact of their HIDS implementation. The solution collects system calls type and arguments, and analyzes traces with a Marklov model. The authors train the model with IDEVAL dataset and some locally generated data. They admit their solution adds a noticeable time overhead, which though does not significantly impact the overall time performance of the system. [14] provides a more precise focus on timing impact of HIDS deployment on system time performance. They show a large gap of time overhead, from 3% to 77%, depending on the type of executed program. However, data are not well representative for our target system. Firstly, the study is from 2004: hardware has evolved and so time performance did. Secondly in the authors' implementation, the monitored workload is a virtual machine hosted by a type-II hypervisor. A type-II hypervisor runs inside an OS; it requires high computing power. This system architecture is not particularly adapted for embedded systems. For our mixed-criticality system instead, we use a type-I hypervisor, which directly manages hardware resources.

In our literature research, all machine-learning based HIDS implementations we found are dedicated for deployment for generic computers running rich operating systems such as Linux or Windows. For instance, the main open-source datasets correspond to trace of execution on desktop PC: e.g. ADFA-LD [4, 11] dataset on Linux OS and KDD99 [19] on Solaris OS.

3 Methodology

We propose a methodology to evaluate the impact of intrusion detection based on system call monitoring on a mixed-criticality system with real-time constraints. The required system call monitoring causes an additional delay in the kernel at system call handling, which induces two consequences: an impact on workload performance as well as an impact on system security.

3.1 Impact on Performance

A mixed criticality system is composed of several workloads corresponding to diverse criticality levels. We distinguish two main types of workload:

- critical: software under deterministic constraints. E.g. Avionics control loops.
- non-critical: software with no deadlines. E.g. Linux OS.

3.1.1 Critical Workload

To use system call tracing in a critical workload, the additional overhead should be added to Worst Case Execution Time (WCET) of this workload. The additional delay due to system call tracing is easy to identify. We can indeed calculate the maximal overhead duration with WCET analysis; the precise tracing overhead can be estimated for each system call in the kernel. Thus, we can compute the worst case overhead, corresponding to the worst system calls combination for the analyzed software.

3.1.2 Non Critical Workload

WCET analysis does not make much sense for non-critical workloads such as Linux OS: it would either be too pessimistic or non doable. We instead aim to analyze an average time overhead. However, because of the large diversity of software, it is not possible to get a representative single average value.

3.2 Impact on System Security and Dependability

In addition to the performance impact, the system call tracing time overhead located in the kernel can potentially compromise the whole system, by breaking time isolation between workloads. A workload raising a huge amount of system calls induces indeed a non-negligible delay in the kernel.

Depending on the kernel implementation, we consider two scenarios:

1. system call handling can be preempted by context switch.
2. system call handling can not be preempted by context switch.

In the first case, even though the application induces a huge delay due to system call tracing, it is stopped by the tick interrupt, responsible for context switching; the system call is possibly aborted. Thus, time isolation is respected.

In the second case, the application can extend slightly its time window, at the expense of other workloads: time isolation is not respected anymore. An example of such issue consists in two applications running on the same single CPU core. The first is a non-critical workload, while the second is a hard real-time workload with firm deadlines. If the non-critical application runs huge amount of system calls, there are high chances that context switch tick interrupt is raised during system call handling. The kernel waits for system call to return to the application before switching context to the next workload. Thus, we can assess the maximal time overhead per context-switch to be the length of system call handling, including system call tracing. As the system is designed for guaranteeing time isolation without tracing, the time window of the non-critical application can be extended to a maximum of $t_{overhead_{max}}$. After a certain amount of context switches, the critical application possibly fails missing its deadline. Thus, we need to define the overall time overhead induced by system call instrumentation for typical workloads, to then be able to evaluate how it could compromise the system.

3.3 Estimating the Tracing Time Overhead

We propose to compute an average time overhead at runtime for a set of representative non-critical workloads, with the following approach. The total time overhead t of an application running a sequence of N system calls corresponds to $t = \sum_{i=1}^{N} t_{syscall_i}$. The quantity t depends on the number of system calls and on their type. Firstly, we count N the amount of system call traces for every workload. Depending on the context of execution, the quantity N can vary for two different executions. Secondly for a set of well used system calls, we compute the time overhead caused by tracing. We can expect a small variation of tracing time between different system calls. The difference is indeed due to variations in the quantity of traced data caused by the number of parameters and return value. Finally, we can estimate a range of possible values for the total tracing time overhead t, so that $t_{min} \leq t \leq t_{max}$ with $t_{min} = \frac{t_{tracing_{min}}}{T} * N$ and $t_{max} = \frac{t_{tracing_{max}}}{T} * N$. T is the total time of execution, without tracing. We can not precisely measure this value: program execution varies, depending on the context of execution and regardless of system call tracing feature. Therefore for a single workload execution, system call tracing time must be associated with total execution time. Instead of T, we can measure T' so that $T' = T + t$. To finally evaluate the impact of system call tracing, we will use the ratio $r = \frac{t}{T'-t}$.

4 Experiment

4.1 System Design

4.1.1 Overview

The figure 1 details the system architecture. We run the experiment on PikeOS [8], a commercial certified mixed-criticality OS. It can be used as a real-time OS (RTOS) to run native applications. As a type-I hypervisor, it can also host more complex workloads such as complete OS. Every hosted workload is attributed levels of safety and security, where resource partitioning enforces isolated execution. A diverse range of workload types is supported (ARINC 653, Linux, POSIX, AUTOSAR, etc.). PikeOS software complies with safety standards, such as DO-178B (Avionics), EN 50128 (Railways).

The system call tracing driver stores system call traces in a per CPU core trace buffer and returns them upon request. For each system call raised, two traces are stored. They correspond to the entry and exit hooks inserted in every system call handler. Once a system call is received, the entry hook extracts the system call type and its parameters and writes the first trace into the corresponding trace buffer. Before returning from the handler, the exit hook gets the return value and stores the second trace into the trace buffer.

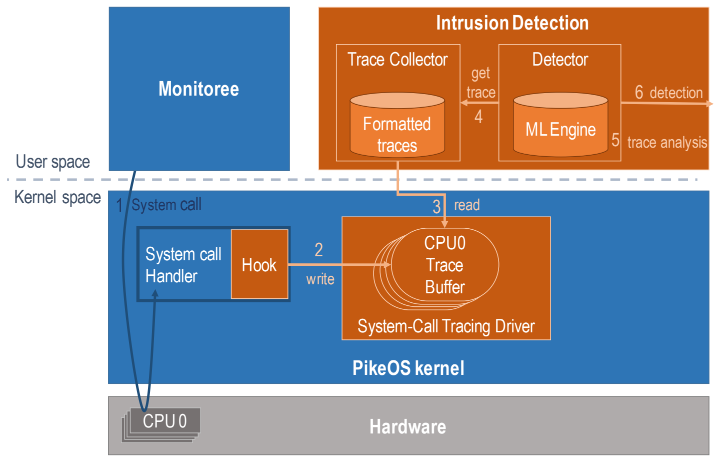

Figure 1 System architecture.

At user-level, the Monitoree, defined in 4.2.2, is the monitored workload. The IDS software is composed of two main user applications. Firstly, the detector contains the analytics model to identify a trace as footprint of an intrusion or as normal activity. This part is not yet implemented and will be developed in further work. Secondly, the trace collector collects traces by communicating with the system call tracing driver. It formats and stores the traces locally until read request from the detector.

4.1.2 System Call Tracing within PikeOS

The system call tracing process goes through the following steps (see figure 1):

1. The monitoree raises a system call.
2. The OS kernel handles the system call. From events raised by hooks in the handler, system call information are stored in the trace buffer of the initiating CPU core.
3. At some point, the trace collector requests traces to the system call tracing driver. The driver reads the trace buffer and returns traces to the trace collector. Traces are then formatted and stored waiting for request from the detector.
4. The detector requests a trace to the trace collector.
5. The trace is analyzed with the machine-learning engine.
6. In the case where an intrusion is detected by the engine, a signal is raised (e.g. an alarm message is printed on the console).

Groups of operations (1,2), (3), (4,5,6) can run in parallel. Operations inside the groups are sequential.

The tracing feature induces an additional delay for system call handling in the kernel. It corresponds to the entry and exit hooks inserted in system call handlers. The hook consists mainly in a memory copy operation into the CPU trace buffer.

For the sake of clarity, we isolate the monitoree on a single dedicated CPU core (core 0). The trace collector runs on a distinct core (core 1).

4.1.3 Limitations of the Implementation

For now, the solution does not monitor hardware virtualized guest OS. Such workload indeed directly handles system calls, without any intervention of the hypervisor. Critical instructions raise hypervisor calls and are trapped in the kernel. Our solution could be adapted to support hardware-virtualized workloads, by tracing these calls in the hypervisor kernel.

4.2 Setup

4.2.1 Test Environment

For the experiment we use the NXP board QorIQ LS1043A. Time is measured with CNTVCT_EL0 CPU core counter. It corresponds to the virtual count for execution level 0 (user mode) of the running core and is accessible from user workloads. The resolution of this counter is 40 ns. A timestamp is defined in the kernel for every trace before it is stored in the trace buffer.

In our tests, we measure the time for running a workload, by printing the counter value just before and after its execution. We get the unitary time for system call tracing in the kernel using the same method.

4.2.2 Tested Applications

We selected diverse applications from native and POSIX applications, as well as paravirtualized Linux. Native and POSIX servers are stimulated with a Linux client running `ping` process. For every test, we run measurements only once the system has booted.

Native applications are C programs running directly on PikeOS, using specific libraries. Ping server is a ICMP echo server which returns echo replies upon request. Shared memory client/server application corresponds to two applications, depending on runtime mode. Both applications share a memory buffer to exchange 32 text messages that are emitted by the server and read by the client, using synchronization mechanisms.

The POSIX application inetd is a server daemon, which relies on a widely used TCP/IP stack called lightweightIP.

The Linux workload corresponds to a paravirtualized commercial embedded Linux for PikeOS, called ElinOS [7]. Our selection gathers well used processes and common performance benchmarks. Table 1 lists command lines for all Linux applications.

Table 1 List of Linux workloads.

Workloads	Command line
unixbench	`unixbench`
dd	`dd if=/dev/zero of=FILE count=CNT bs=BS`
ps	`ps -ef`
find	`find / > /dev/null 2&>1`
netserver	`netserver -4 -L ADDRESS -D -d`
netperf	`netperf -H NETSERVER`
Pacman	`pacman`
gzip	`gzip FILE`
iperf	`iperf -s`
ping	`ping ADDRESS -c 20`

5 Results

5.1 Reliability of Measurements

Every test is reproduced five times. Average values of time and amount of collected system calls traces are represented in table 2. To evaluate the variability of results, the standard deviation is calculated and compared to the average value. Variability of system call count measurements is high in two situations:

- for workloads which raise few system calls; e.g. native ping server and empty Linux OS.
- for short execution time; e.g. ps and dd Linux processes.

In this second category, time measurement is also less precise (approximately 7% of fluctuation).

5.2 System Calls in Workloads Selection

The table 2 shows that the amount of system calls varies, from 23 system calls to more than 200,000 for one second of workload execution, excluding tracing time. The results illustrate that native applications, even with networking, raise very few system calls, comparing to POSIX inetd daemon and Linux workloads.

Table 2 List of measurements for tested workloads.

Workloads		time (s)	calls/s	var (%)
Native	ping server	60.45	23.2	41.16
	ping server (busy)	63.98	552.0	9.60
	sh.mem. (client)	31.47	410.0	0.00
POSIX	inetd (no requests)	30.08	10,508.0	2.75
	inetd (ping requests)	30.06	11,598.2	0.39
Linux	empty	60.34	26.7	8.80
	dd (64 MiB)	0.11	34,000.0	9.57
	dd (128 MiB)	0.22	19,398.1	7.52
	ps -ef	0.01	225,876.8	12.10
	find /	0.31	115,113.0	5.86
	gzip (128 MiB)	3.48	382.0	2.70
	ping (20 packets)	19.03	375.5	5.58
	netserver	10.01	6,120.9	0.74
	netperf	10.02	6,289.6	0.94
	iperf (server)	10.02	6,196.5	0.87
	iperf (client)	10.04	6,170.6	1.11
	pacman	20.00	4,129.5	2.95
	pacman	30.00	4,687.3	0.87
	unixbench	1,677.24	4,799.5	0.14

5.3 Unitary System Call Tracing Overhead

The granularity of the hardware counter does not allow high precision; with the 40 ns threshold, we count less than 20 ticks per unitary tracing time overhead. Nevertheless, because of the high variability in collected values (around 20%), we can still define a range of time overhead for a set of common system calls. We measure tracing overhead for the system

Table 3 List of tracing overhead for a selection of system calls.

System call	Min (ns)	Average (ns)	Max (ns)	var (%)
MMAP	240	391	760	20
IPC	240	399	760	19
Thread Yield	240	343	720	24
Thread Sched.	240	362	640	19
Sleep	240	382	760	23

calls described in table 3. They indeed appear as good candidates for intrusion detection as they have a direct impact on system execution; i.e. workloads scheduling, memory access, inter-thread communication.

- Memory mapping (MMAP): maps a number of pages from one workload's address space to another.
- Inter-process communication (IPC): sends a message to a thread and receives an IPC message from another.
- Thread Yield: forces the running thread to yield the CPU.
- Thread Scheduling: exchanges thread priority and execution time window.
- Sleep: suspends the calling thread for an amount of time.

For every system call, we run 1000 measurements. For a single system call, tracing time can take a wide range of values between 240 ns and 760 ns: minimal, average, and maximal values are respectively equal to 240 ns, 376 ns, and 760 ns.

5.4 Interpretation of Results

Table 4 Time overhead for tested workloads.

Workloads		Time overhead ratio		
		min	average	max
Native	ping server (busy)	1.32E-04	2.08E-04	4.20E-04
	sh.mem. (client)	9.84E-05	1.54E-04	3.12E-04
POSIX	inetd (ping requests)	2.78E-03	4.36E-03	8.81E-03
Linux	empty	6.41E-06	1.00E-05	2.03E-05
	dd (128 MiB)	4.66E-03	7.29E-03	1.47E-02
	ps -ef	5.42E-02	8.49E-02	1.72E-01
	find /	3.63E-03	5.68E-03	1.15E-02
	gzip (128 MiB)	9.17E-05	1.44E-04	2.90E-04
	ping (20 packets)	9.01E-05	1.41E-04	2.85E-04
	netperf	1.51E-03	2.36E-03	4.78E-03
	iperf (server)	1.49E-03	2.33E-03	4.71E-03
	pacman	1.12E-03	1.76E-03	3.56E-03
	unixbench	1.15E-03	1.80E-03	3.65E-03

According to the definition of security impact provided in section 3, we compute the time ratio, $r = \frac{t_{tracing}}{T_{tot} - t_{tracing}}$. $t_{tracing}$ is in the range $[t_{min}; t_{max}]$. Let $t_{tracing}$ be a variable so that $r = f(t_{tracing})$. r is an increasing function between $\frac{t_{min}}{T_{tot} - t_{min}}$ and $\frac{t_{max}}{T_{tot} - t_{max}}$. We show minimal, average, and maximal values of r in table 4.

r values can vary significantly in function of the workload from 10^{-5} to 17%. Thus, we can not easily define a factor to predict time overhead in function of simple parameters such as time of execution and amount of static code instructions. The time overhead depends on the type of system calls and context of execution (especially for Linux virtual machines).

As seen in section 5.2, because the amount of system calls for native applications is very low, the time overhead is negligible. More complex workloads, e.g. Linux and POSIX programs, present much higher time overhead. `ps` Linux process spends particularly the longest time for tracing system calls (5% to 17% of additional time). The majority of the selected workloads correspond to a ratio r in the order of 10^{-3}, which represents a tracing time overhead of 60 ms for one minute of program execution time. The overhead affects workloads time performance, but remains reasonable in the majority of cases.

6 Discussions

6.1 Comparison of Tracing Impact with Previous Work

[14] implements a HIDS for monitoring a Linux virtual machine running on a type II hypervisor called User-Mode Linux (UML) Monitor. The authors compute total time overhead of system call monitoring for three well used Linux processes: ps, find, and ls. The overhead corresponds to the whole implementation, system call tracing and detection. Their results point already a wide diversity of time overhead, from 3% with `ps -ef` to 77% for `find / > /dev/null 2&>1`. Our implementation shows a much reduced overhead: average values are 8% in the first case and less than 1% for the second operation.

Another approach [16] analyzes the occurrences of system calls in workloads. From the high throughput of system calls processed by their solution, the authors deduce a visible but negligible impact at runtime: they argue that their HIDS can process from 12000 to 22000 system calls per seconds when common operating system's services usually run around 2000 system calls per second (six to ten times lower). This argument seems questionable for modern virtual machines, in light of our results; in our test, many of the applications raise more than 4000 system calls per second (Pacman, dd, netserver, etc.).

6.2 Security Impact

As highlighted in section 3.2, in the case where the kernel implementation can not preempt system call handling, time isolation is compromised in the system. According to our measurements, a workload can extend its time window to maximum $t_{overhead_{max}} = 760$ ns between two context switches.

6.2.1 Workloads Context-Switch Rate Influence

The risk of system call tracing impact on system security increases by reducing the context switch period. The shorter the time period for an application running system calls is, the more probable a context switch to be delayed by system call tracing is. Table 5 shows an example of worst case execution, where every context switch is delayed by a system call. The system runs two workloads, one with real-time constraints and a non-critical workload raising continuously system calls. This second software corresponds to a Linux thread repeating the command `ps -ef` in an infinite loop. In function of the context switch rate, the table shows average amount of system calls raised by the monitored application between two context switches and the final tracing delay for 1 s of non-critical workload cumulated execution.

Table 5 Influence of the context switch period on tracing overhead.

Context Switch Rate	Call/period	Overhead (1s execution)
100 μs	2.26E-04	172 ms
100 ns	2.26	8 μs

This vulnerability could be exploited by an attacker running exclusively system calls from a non-criticalprogramd to compromise availability of a critical program.

6.2.2 PikeOS Real-Time Hypervisor Use-Case

PikeOS kernel is non-preemptive. Thus, context switch can not directly preempt system call handling in the kernel. Nevertheless, to control time spent in the kernel, it includes preemption points, notably for long operations and before returning to user-space. Some long system call handlers, such as creating or destroying a task (above 10 μs in average), can include preemption points; though most of them do not, because of their speed (generally less than 1 μs). Hence, tracing impact described in section 6.2.1 also applies to PikeOS real-time hypervisor.

6.3 Considerations for the Detection Engine Implementation

Intrusions in mixed-criticality systems most likely target non-critical workloads: unlike closed critical workloads, they usually contain vulnerabilities. The attacker aims either to steal data from the system or to disturb its execution. In both cases, she eventually has to interact with the system (OS kernel, other workloads) from the non-critical software. Thus, our strategy consists in monitoring all workloads using a single detector. The detector would analyze system call generic information (parameters, type, etc.), correlated to the thread 's criticality level and its execution context: e.g. the running CPU core, and interactions with other workloads. As soon as a thread is identified as a threat, it should be isolated from other system entities. We also consider rollback mechanisms, especially for affected critical workloads to restore them in a safe state.

6.4 Other Promising Features for ML-based HIDS

To detect intrusions in a mixed-criticality system with Machine-Learning, we also consider further interesting features.

6.4.1 Performance Counters

Performance counters tracing seems a promising approach to detect intrusions with machine-learning, as highlighted by [1] and [6]. Tracing performance counters's values would have no direct impact on system real-time constraints as they can be read from user mode, without kernel intervention. Tracing can then be done in a separate monitor workload, running on same CPU core as the monitored workload, subject to the compliance of the monitor with system real-time constraints. This must be ensured, as well as for every workload, by the OS kernel.

6.4.2 Branch Tracing

Branch instructions provide relevant information regarding security, e.g. to guarantee program integrity with CFI (see section 2). Although we could not find any publication on this topic, we consider branch instructions as interesting features to investigate for machine-learning based HIDS. Tracing branch transfers can be done either in OS kernel software or in the hardware: e.g. ARM CoreSight [3] debugging hardware provides this tracing feature. Hardware implementation is preferable for performance speed-up and time isolation of workloads. With appropriate access rights, the monitoring application can indeed collect traces directly from user mode, without systematic interactions with the kernel.

6.4.3 Network Monitoring

Research of the last two decades has well investigated network intrusion detection systems (NIDS), that are even proposed in main IT security solutions on the market. Network monitoring traces network packets transiting from and to the monitored application. It relies on same principle of system call tracing, updating the tracing location from system call handler to the network driver. As well as with system call tracing, the overall timing impact should be non negligible.

7 Conclusion

We evaluated the impact of system call instrumentation for deploying intrusion detection in embedded mixed-criticality systems, through a concrete implementation based on PikeOS hypervisor. The implementation shows a reasonable time overhead for the majority of workloads. However, because of new delays induced for tracing in the OS kernel, time isolation between workloads is not guaranteed anymore. In further work, we plan to develop a system call based HIDS to monitor a mixed-criticality system, which enforces time isolation. We will research adapted HIDS architectures for our system context. We will investigate how system call flows from different workloads can be combined for a reliable detection.

References

1 Muhamed Fauzi Bin Abbas, Sai Praveen Kadiyala, Alok Prakash, Thambipillai Srikanthan, and Yan Lin Aung. Hardware Performance Counters Based Runtime Anomaly Detection Using SVM. In *TRON Symposium (TRONSHOW)*, 2017.

2 Jim Alves-Foss, W. Scott Harrison, Paul Oman, and Carol Taylor. The MILS Architecture for High-Assurance Embedded Systems. In *International Journal of Embedded Systems*, 2005.

3 ARM. *CoreSight Technical Introduction*, 2013. White Paper: ARM-EPM-039795.

4 Gideon Creech and Jiankun Hu. Generation of a New IDS Test Dataset: Time to Retire the KDD Collection. In *IEEE Wireless Communications and Networking Conference*, 2013.

5 M. T. Elgraini, N. Assem, and T. Rachidi. Host intrusion detection for long stealthy system call sequences. In *Colloquium in Information Science and Technology*, 2012.

6 David Fiser and William Gamazo Sanchez. Detecting Attacks that Exploit Meltdown and Spectre with Performance Counters. `https://blog.trendmicro.com/trendlabs-security-intelligence/detecting-attacks-that-exploit-meltdown-and-spectre-with-performance-counters/`, 2018. [Jun. 05, 2019].

7 SYSGO GmbH. ElinOS Embedded Linux Webpage. `https://www.sysgo.com/products/elinos-embedded-linux/`. [Jun. 05, 2019].

8 SYSGO GmbH. PikeOS Hypervisor Webpage. `https://www.sysgo.com/products/pikeos-hypervisor/`. [Jun. 05, 2019].

9 W. Haider, J. Hu, , and M. Xie. Towards reliable data feature retrieval and decision engine in host-based anomaly detection systems. In *IEEE 10th Conference on Industrial Electronics and Applications (ICIEA)*, 2015.

10 Mohamed Hassan. Heterogeneous MPSoCs for Mixed CriticalitySystems: Challenges and Opportunities. In *IEEE Design and Test Magazine*, 2017.

11 Jiankun Hu. AFDA-LD dataset Webpage. `https://www.unsw.adfa.edu.au/unsw-canberra-cyber/cybersecurity/ADFA-IDS-Datasets/`, 2013. [Jun. 05, 2019].

12 Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

13 Koucham, T. Rachidi, and N. Assem. Host intrusion detection using system call argument-based clustering combined with Bayesian classification. In *SAI Intelligent Systems Conference (IntelliSys)*, 2015.

14 M. Laureano, C. Maziero, and E. Jamhour. Intrusion detection in virtual machine environments. In *30th Euromicro Conference*, 2004.

15 Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, A. Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

16 F. Maggi, M. Matteucci, and S. Zanero. Detecting Intrusions through System Call Sequence and Argument Analysis. In *IEEE Transactions on Dependable and Secure Computing*, 2010.

17 Lee Pike, Pat Hickey, Trevor Elliott, Eric Mertens, and Aaron Tomb. TrackOS: A Security-Aware Real-Time Operating System. In *International Conference on Runtime Verification*, 2017.

18 The LLVM Foundation. The LLVM compiler infrastructure. `llvm.org`. [Jun. 05, 2019].

19 University of California. KDD Cup 1999 Data Webpage. `https://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`, 1999.

20 C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In *IEEE Symposium on Security and Privacy*, 1999.

Argument Patterns for Multi-Concern Assurance of Connected Automated Driving Systems

Fredrik Warg
RISE Research Institutes of Sweden, Borås, Sweden
http://www.ri.se
fredrik.warg@ri.se

Martin Skoglund
RISE Research Institutes of Sweden, Borås, Sweden
martin.skoglund@ri.se

Abstract

Showing that dependable embedded systems fulfil vital quality attributes, e.g. by conforming to relevant standards, can be challenging. For emerging and increasingly complex functions, such as connected automated driving (CAD), there is also a need to ensure that attributes such as safety, cybersecurity, and availability are fulfilled simultaneously. Furthermore, such systems are often designed using existing parts, including 3rd party components, which must be included in the quality assurance. This paper discusses how to structure the argument at the core of an assurance case taking these considerations into account, and proposes patterns to aid in this task. The patterns are applied in a case study with an example automotive function. While the aim has primarily been safety and security assurance of CAD, their generic nature make the patterns relevant for multi-concern assurance in general.

2012 ACM Subject Classification Computer systems organization → Dependable and fault-tolerant systems and networks; Computer systems organization → Embedded and cyber-physical systems

Keywords and phrases Multi-concern assurance, connected automated driving, dependability, functional safety, cybersecurity, cyber-physical systems, critical embedded systems

Digital Object Identifier 10.4230/OASIcs.CERTS.2019.3

Funding This work was supported by Vinnova – project ESPLANADE (2016-04268), and the EU and Vinnova – project ECSEL AMASS (692474), but the contents of the paper only reflect the authors' views.

1 Introduction

When releasing embedded dependability-critical electrical/electronic (E/E) systems on the market it is typically necessary to demonstrate that they are sufficiently safe. This is often done by showing compliance to a functional safety standard. A key design strategy to successfully show the product is safe is to keep the safety-related part of the system as small and simple as possible. While this is still desirable, the technological development is inexorably moving towards higher complexity even for safety-related functionality. One example is automated driving systems (ADS) [17], i.e. functions enabling what is commonly referred to as self-driving or autonomous vehicles. For such functions it is difficult to keep the safety-related part small and isolated as the goal of the function is to drive safely, a task which by necessity involves many of the vehicle's sensory, control and actuator subsystems.

With this complexity, it becomes more difficult to convincingly show that a product is safe. A further complication is that safety cannot be treated in isolation in the presence of other quality attributes (QAs) that may affect safety properties. For instance, it is expected that most ADS equipped vehicles are also connected in order to increase performance of the functionality, e.g. an ADS that exchanges information with surrounding vehicles and roadside infrastructure (traffic lights, signs, roadside sensors) will have a better world model and be able to make better and less defensive driving decisions. However, adding connectivity to enable connected automated driving (CAD) also increases the security risks. A hacker may compromise the ADS remotely to e.g. circumvent its safety mechanisms. Hence, to demonstrate that the function is safe it is also necessary to show that all security risk that may compromise safety have been treated. This extends to arbitrarily many interrelated concerns, e.g. it might be necessary to consider availability of the function to make sure safety and security mechanisms do not lower availability of the function in a way that makes the business case unviable.

Safety for vehicle E/E functions is typically demonstrated by showing conformance to the ISO 26262 standard [8], through a safety case consisting of artefacts resulting from complying to all its normative requirements. The implicit argument is that standard conformance itself is proof of safety. However, making an explicit argument showing the product-specific safety-rationale within the overall framework of the standard can aid both development engineers and safety assessor [3]. For this work our premise is that such an argument is even more important - even if the standards may not mandate it - when the complexity increases, for instance when showing simultaneous conformance to several standards each representing a different QA, also called *multi-concern assurance*, or when including components developed out-of-context by 3rd party suppliers.

Our contribution in this paper is a discussion and proposed patterns for simultaneously covering the dimensions of *(1) multiple concerns*, *(2) standards conformance*, *(3) element-out-of-context*, and *(4) system lifecycle* in an argument for a multi-concern assurance case. A danger with such multi-dimensional arguments is that it becomes unwieldy and incomprehensible, thus failing to fill its basic purpose, something we attempt to overcome with a clear and regular structure. We also develop such an argument in a case study relevant for CAD.

In work related to ours, others have suggested patterns for ISO 26262 arguments [4, 5, 7, 13, 15, 16]. Most of these provide more detailed patterns that could be combined with what we are proposing, but in some cases also have a somewhat different way of organizing the argument. However these are for safety only. Taguchi et al. [19] discuss and show patterns for different ways of integrating safety and security, which may be done by combining the two concerns, treating them totally separately, or by handling interdependencies in different ways. In this work we use what Taguchi calls a bi-directional reference process pattern as a multi-concern pattern, but also combine it with the other dimensions we discuss. Work focused on co-engineering and how to capture trade-offs between concerns in the argumentation has also been done [1, 12, 6]. In contrast our work is about structuring the assurance case to capture information about inter-concern dependencies together with the other mentioned argument dimensions, not how to handle trade-offs or co-engineering.

2 Argument Dimensions

Here we elaborate on the implication of taking the four dimensions mentioned in Sec. 1 into account. As we focus on dependability aspects and standards conformance, the conceptual V-model used in e.g. many functional safety standards is used to illustrate the lifecycle. In Fig. 1, a triple V-model highlighting the aspects of nominal function, safety and cybersecurity is shown. While some standards and work processes prefer other models, e.g. to highlight iterative aspects more clearly, we note that interpreted as a dependency chain rather than a timeline, the concepts expressed in the V-model are usually applicable, i.e. there is an overall function concept which is refined to implementable components, and tested in several integration levels. When a version of the product is completed, the process is repeated to add new functionality for the next version, while the current version goes into production and maintenance phases. In functional safety standards such as ISO 26262 [8], results from all design and verification steps are collected in a *safety case*, which must be complete and consistent for each product version. When treating several concerns the general term *assurance case* is used instead. In this section we discuss the rationale behind the four dimensions and return to the patterns in the next section.

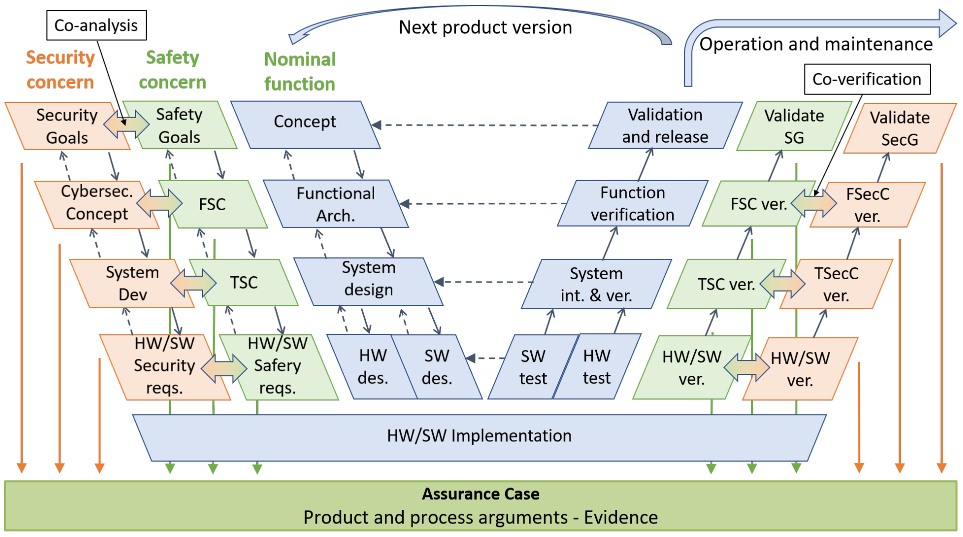

Figure 1 V-model with safety and security attributes and a multi-concern assurance case.

2.1 Lifecycle

As standards and/or company-specific work processes typically prescribe specific development lifecycles, making the lifecycle stages evident in the argument makes it easier to relate the argument to the design process, and thereby show that the risk of introducing systematic faults is sufficiently reduced throughout the lifecycle. In other words, showing the lifecycle in the argument reduces the risks due to misunderstandings and omissions originating from bad mapping between argument and the actual development work. Furthermore there is often a distinction of product and process arguments in standards. Process arguments also include e.g. management practices that are not specifically tied to the product. In our pattern we make a separation of these for increased clarity.

2.2 Standards Conformance

The argument should also preferably reflect if the assurance case shall show conformance to a standard (this is also called a *conformance case*). Our patterns are designed to be compatible with the V-model used in e.g. many functional safety standards. While the patterns create the general structure for the argument, the claims must be instantiated for the specific quality attribute and supported by more low-level claims and supporting evidence. These sub-claims can in many cases be requirements directly from the standard. Thus conformance is not a separate pattern, rather the phases and modules used in our patterns are suitable for combining with standard requirements. Using a tool allowing compliance mapping between standard requirements and the argument, e.g. OpenCert [14], it is even possible to track that all standard requirements have been fulfilled within the framework of the argument.

2.3 Concerns and Their Interplay

If two quality attributes are independent, i.e. fulfilling one can never impact the other, the only aspect of multi-concern assurance compared to separate conformance to the concerns is possible synergies to reduce the assurance work, e.g. joint testing using the same test frameworks (co-verification in Fig. 1). However, typically interplay in the form of potential dependencies, conflicts or synergies between the concerns exist and must be identified and resolved in the multi-concern argument. Again, the analysis of interplay between concerns must cover the entire product lifecycle to make sure conflicts do not appear in any design stage or even after production. For instance, security concerns may make over-the-air (OTA) updates necessary so that security holes uncovered long after the product is released can be fixed, but this makes it necessary to make sure OTA updates cannot compromise safety. We therefore include argument for interplay in our patterns. It should be noted that interplay arguments can become unwieldy if many dependent concerns are treated since the possible combinations quickly grows with number of QAs. Based on [11] as well as own experience, we also claim that specifying well-defined interaction points between concerns (co-analysis in Fig. 1) is preferable to processes integrating several concerns.

2.4 Component Structure

In many domains, including automotive, the most common way to build new features is to integrate parts from suppliers, or reusing existing components. These must be included in the assurance case for the new feature. A supplier may also sell the same part to several customers and even develop it before having any requirements from an OEM. The supplier may then construct their own assurance case for their part, using assumptions on its use, i.e. an assumed context. In ISO 26262, this is called a safety-element-out-of-context. We generalize the concept and use the term element-out-of-context (EooC), which may have an assurance case for multiple concerns. When integrating the EooC in the complete feature, there must be a bridge between the feature and EooC assurance cases explaining how the part developed for the assumed context will also fulfil the requirements for the actual feature in the real context.

3 Putting it Together in Argument Patterns

3.1 Pattern Notation

We use the argumentation structure defined in ISO/IEC 15026-2:2011 [9] which is illustrated on the left hand side of Fig. 2. In this standard, an argument consists of one or more top-level *claims* supported by sub-claims, *evidence*, and/or *assumptions* through an *argument* detailing how these underlying components support the top-level claim. Sub-claims must be supported in the same way; the argument can be arbitrarily many levels with evidence and assumptions as leaf nodes. The choice of top-level claims must be supported by a *justification*, as must an argument (at any level) have a justification for how underlying components support a (sub-)claim. The standard is agnostic as to how the arguments are represented. In this paper we use the GSN notation [18], which provides an illustrative graphical representation, to show our proposed patterns. The GSN notation corresponding to the 15026-2 concepts are shown to the right in Fig. 2; in GSN a claim is called a *goal*, an argument is called *strategy* and evidence is called a *solution*.

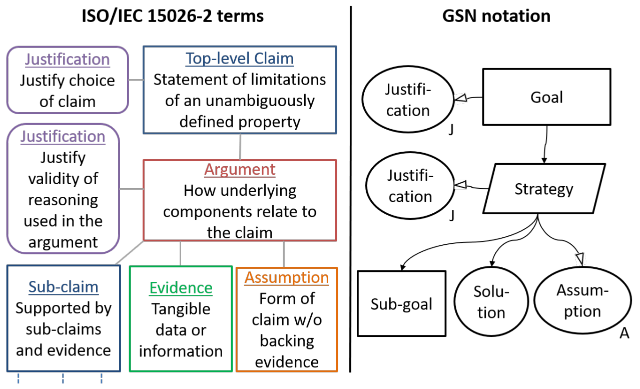
ISO/IEC 15026-2 terms
GSN notation
Justification
Justify choice of claim
Top-level Claim
Statement of limitations of an unambiguously defined property
Justification
Justify validity of reasoning used in the argument
Argument
How underlying components relate to the claim
Sub-claim
Supported by sub-claims and evidence
Evidence
Tangible data or information
Assumption
Form of claim w/o backing evidence
Justifi-cation
J
Goal
Justifi-cation
J
Strategy
Sub-goal
Solu-tion
Assum-ption
A

Figure 2 ISO/IEC 15026-2:2011 argument and corresponding GSN notation.

In addition to the basic notation we use some extensions to GSN. The modular extensions are helpful for managing the complexity of large arguments, and the extensions allowing for abstraction are used to express generic argument patterns. Fig. 3 shows the GSN elements used in this paper. *(a) Module* is used to represent a separate sub-argument which is used either in a *module view* which is a special overview diagram in GSN showing only relationships between such modules, or to show that a goal is supported by an entire argument contained in a separate module. *(b) Contract* is used when a goal will be supported in a yet unspecified module and is used to provide decoupling. The contract module itself is used to provide a glue argument showing how the argument in a module (which might e.g. be provided for a re-usable component) fulfils the goal which was to be supported by the contract. *(c) Away goal* repeats a goal made in another module in the argument of a local module in order to show dependencies between goals in different modules. The away goal also identifies the module where the original goal can be found. The *(d) InContextOf arrow* is used in a way proposed by the AMASS project [2], which is to show that fulfillment of a goal in one concern is dependent on fulfillment of a goal in another concern. *(e) Option* is used to denote alternatives to satisfy a relationship, while *(f) optional arrow* is used for an optional relationship, and *(g) many arrow* denotes a one-to-many relationship with the cardinality shown next to the arrow. A goal can also be *(h) uninstantiated* which means it is an abstract element that needs to be replaced by a concrete instance. Words within {brackets} in the argument are tokens to be instantiated, e.g. *{Goal}* could be instantiated as *LaneKeepingAssist is acceptably safe* as a top-level goal in the safety argument for a lane keeping assist vehicle feature. *(i) Undeveloped* means a goal which is not yet fully supported, i.e. it needs to be developed by completing the argument beneath it. Goals can be both undeveloped and uninstantiated at the same time. Finally, *(j) context* is part of the basic GSN notation and is used to provide contextual information needed for interpreting the goal or strategy it is attached to. These concepts are more fully described in the GSN community standard version 2 [18]. All GSN figures in the rest of the paper are produced with the OpenCert tool [14][1].

[1] Some modifications of the figures produced by the tool have been made for improved readability.

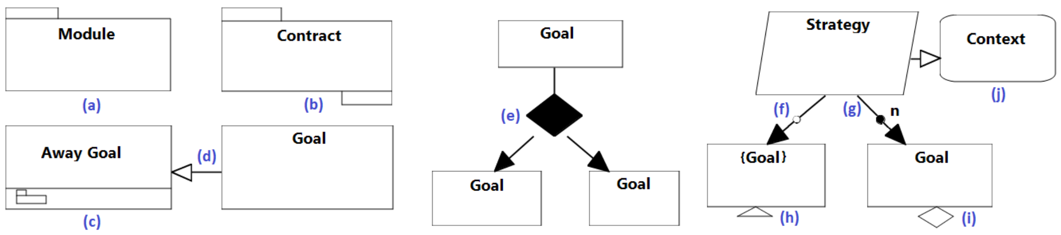

Figure 3 GSN extensions used in the paper.

3.2 Overall Argument Structure and Lifecycle

We organize the overall argument as shown in the GSN modules view in Fig. 4. The top level claim will be that the system (or EooC which we return to in Sec. 3.5) fulfils all quality attributes that have been defined for it. A pattern for the topmost module of Fig. 4 is shown in Fig. 5; this pattern references modules for all QAs and interplay arguments relevant for the product. The concept phase is where initial concept (e.g. item definition in ISO 26262) is defined and risk evaluation is performed (e.g. hazard analysis & risk assessment (HA&RA) and definition of safety goals in ISO 26262). In the concept phase there will be separate modules for each QA and for interplay between all QAs where relevant, e.g. safety and security are not independent and therefore should have an interplay module if they are two of the defined QAs. The rest of the argument is organized according to lifecycle with one module per logical element on the functional concept stage, one module per component on the technical concept/system design stage and modules for software and hardware development for each component. There are separate modules to deal with management and post-development issues such as production, maintenance and decommissioning. These stages are typical for a V-model. The number of abstraction levels may vary but is easy to adapt as the basic structure in each level is the same.

3.3 Concerns

For each quality attribute, a pattern for developing this concern in the concept phase is shown in Fig. 6. The strategy is to use a specified lifecycle, often from a standard, with adequate measures for the QA. The sub-goals are optional depending on the QA, but typical components of the argument is: risk mitigation by using an analysis method and introducing requirements specifying the risks to be avoided (and in many standards the level of risk reduction is quantified with an integrity level [10]); adequate management and operations and maintenance (OaM) practices; and confirmation measures, e.g. review of analysis results.

Following the goal *{QA} requirements introduced to reduce {QA} risks* from one of the leaf nodes in Fig. 6 is a pattern, shown in Fig. 7, for making sure these quality attribute requirements have also been correctly implemented. This pattern provides a way to create a rationale around each QA requirement showing that it has been correctly refined, implemented and verified. The pattern contains goals for refining the QA requirements to the next abstraction level where the pattern will be repeated again for all refined requirements. The refinement goals are optional as they are obviously not applicable on the lowest refinement level while the verified goal is applicable (and mandatory) on all levels.

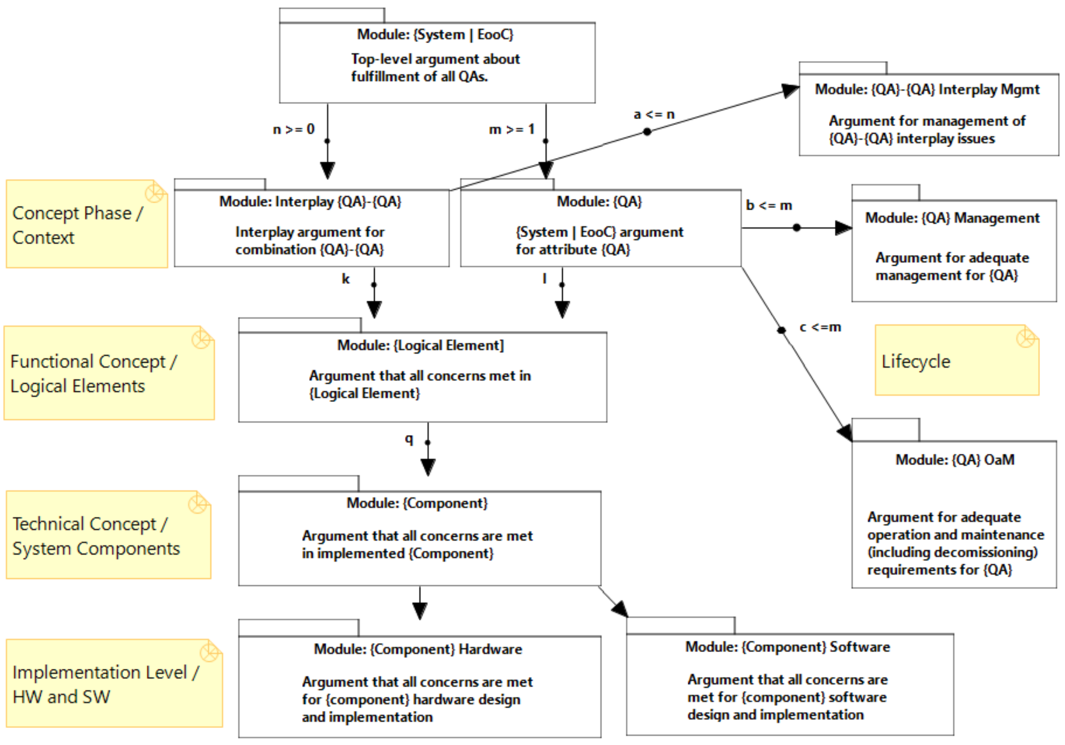
Module: {System | EooC}
Top-level argument about fulfillment of all QAs.
n >= 0
m >= 1
a <= n
Module: {QA}-{QA} Interplay Mgmt
Argument for management of {QA}-{QA} interplay issues
Concept Phase / Context
Module: Interplay {QA}-{QA}
Interplay argument for combination {QA}-{QA}
Module: {QA}
{System | EooC} argument for attribute {QA}
b <= m
Module: {QA} Management
Argument for adequate management for {QA}
k
l
c <=m
Functional Concept / Logical Elements
Module: {Logical Element}
Argument that all concerns met in {Logical Element}
Lifecycle
q
Module: {QA} OaM
Argument for adequate operation and maintenance (including decomissioning) requirements for {QA}
Technical Concept / System Components
Module: {Component}
Argument that all concerns are met in implemented {Component}
Implementation Level / HW and SW
Module: {Component} Hardware
Argument that all concerns are met for {component} hardware design and implementation
Module: {Component} Software
Argument that all concerns are met for {component} software design and implementation

Figure 4 Modules view of system or element-out-of-context.

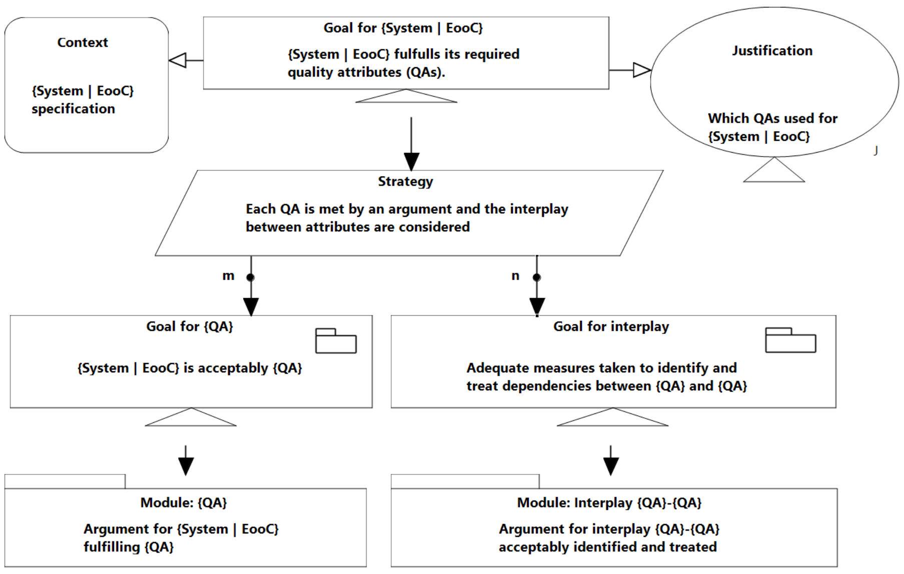
Context
{System | EooC} specification
Goal for {System | EooC}
{System | EooC} fulfills its required quality attributes (QAs).
Justification
Which QAs used for {System | EooC}
J
Strategy
Each QA is met by an argument and the interplay between attributes are considered
m
n
Goal for {QA}
{System | EooC} is acceptably {QA}
Goal for interplay
Adequate measures taken to identify and treat dependencies between {QA} and {QA}
Module: {QA}
Argument for {System | EooC} fulfilling {QA}
Module: Interplay {QA}-{QA}
Argument for interplay {QA}-{QA} acceptably identified and treated

Figure 5 Pattern for top-level multi-concern argument.

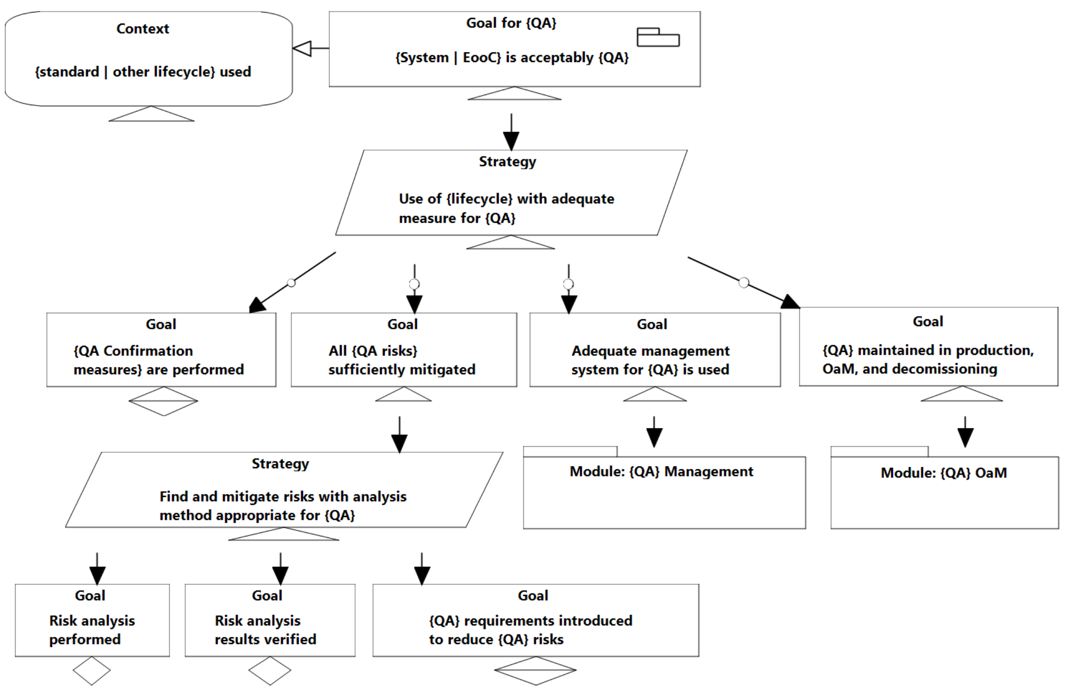
Context
{standard | other lifecycle} used
Goal for {QA}
{System | EooC} is acceptably {QA}
Strategy
Use of {lifecycle} with adequate measure for {QA}
Goal
{QA Confirmation measures} are performed
Goal
All {QA risks} sufficiently mitigated
Goal
Adequate management system for {QA} is used
Goal
{QA} maintained in production, OaM, and decomissioning
Strategy
Find and mitigate risks with analysis method appropriate for {QA}
Module: {QA} Management
Module: {QA} OaM
Goal
Risk analysis performed
Goal
Risk analysis results verified
Goal
{QA} requirements introduced to reduce {QA} risks

Figure 6 Pattern for a quality attribute.

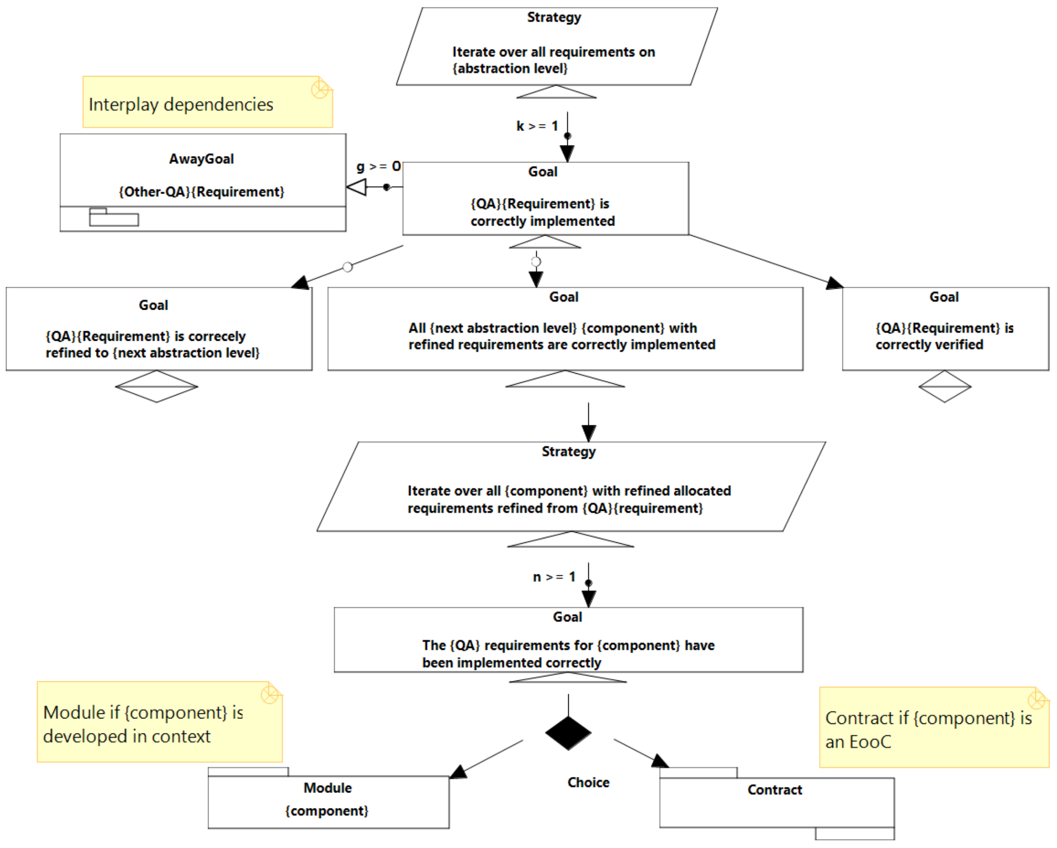
Strategy
Iterate over all requirements on {abstraction level}
Interplay dependencies
k >= 1
AwayGoal
{Other-QA}{Requirement}
g >= 0
Goal
{QA}{Requirement} is correctly implemented
Goal
{QA}{Requirement} is correcely refined to {next abstraction level}
Goal
All {next abstraction level} {component} with refined requirements are correctly implemented
Goal
{QA}{Requirement} is correctly verified
Strategy
Iterate over all {component} with refined allocated requirements refined from {QA}{requirement}
n >= 1
Goal
The {QA} requirements for {component} have been implemented correctly
Module if {component} is developed in context
Contract if {component} is an EooC
Choice
Module
{component}
Contract

Figure 7 Pattern for a QA requirement at any abstraction level.

3.4 Interplay

Interplay can be argued between two or more QAs in each interplay module depending on which methods are found most suitable for interplay in each case. However, it must be evident that all relevant combinations of QAs are taken into account. The pattern, shown in Fig. 8, establishes that management practices for interplay are in place and that dependencies between QAs are found and introduced in the QA requirement hierarchy. This was shown in Fig. 7 as an away goal for a requirement, indicating an interplay dependency. Similar to how QA requirements were handled, the pattern in Fig. 9 then makes sure these interplay dependencies are also refined in lower abstraction levels of the design.

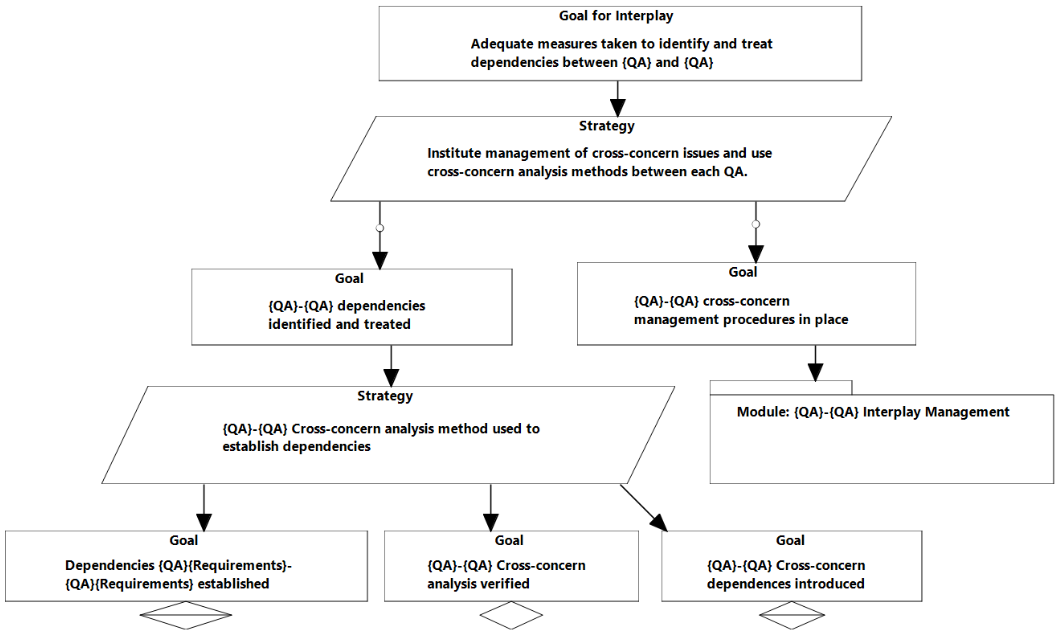
Goal for Interplay
Adequate measures taken to identify and treat dependencies between {QA} and {QA}
Strategy
Institute management of cross-concern issues and use cross-concern analysis methods between each QA.
Goal
{QA}-{QA} dependencies identified and treated
Goal
{QA}-{QA} cross-concern management procedures in place
Strategy
{QA}-{QA} Cross-concern analysis method used to establish dependencies
Module: {QA}-{QA} Interplay Management
Goal
Dependencies {QA}{Requirements}-{QA}{Requirements} established
Goal
{QA}-{QA} Cross-concern analysis verified
Goal
{QA}-{QA} Cross-concern dependences introduced

Figure 8 Pattern for an interplay.

3.5 Element-out-of-Context

The final pattern is the glue between in-context feature and element-out-of-context mentioned in Sec. 2.4. This pattern, shown in Fig. 10, simply connects in-context QAs with the same QA for the EooC, but establishes that an argument showing their compatibility needs to be developed. An analogous pattern (not shown) can be used for the interplay, i.e. it is also necessary to ascertain that the relevant interplay dependencies are covered in the EooC. A component in any abstraction level can be an EooC, which means the EooC will contain the argument from that abstraction level down.

4 Case Study

As a case study we use a positioning element (PE) for CAD which needs to conform to both functional safety and cybersecurity standards. PE is designed as an element-out-of-context (EooC) and can thus be used for various functions. As it is aimed at the automotive domain,

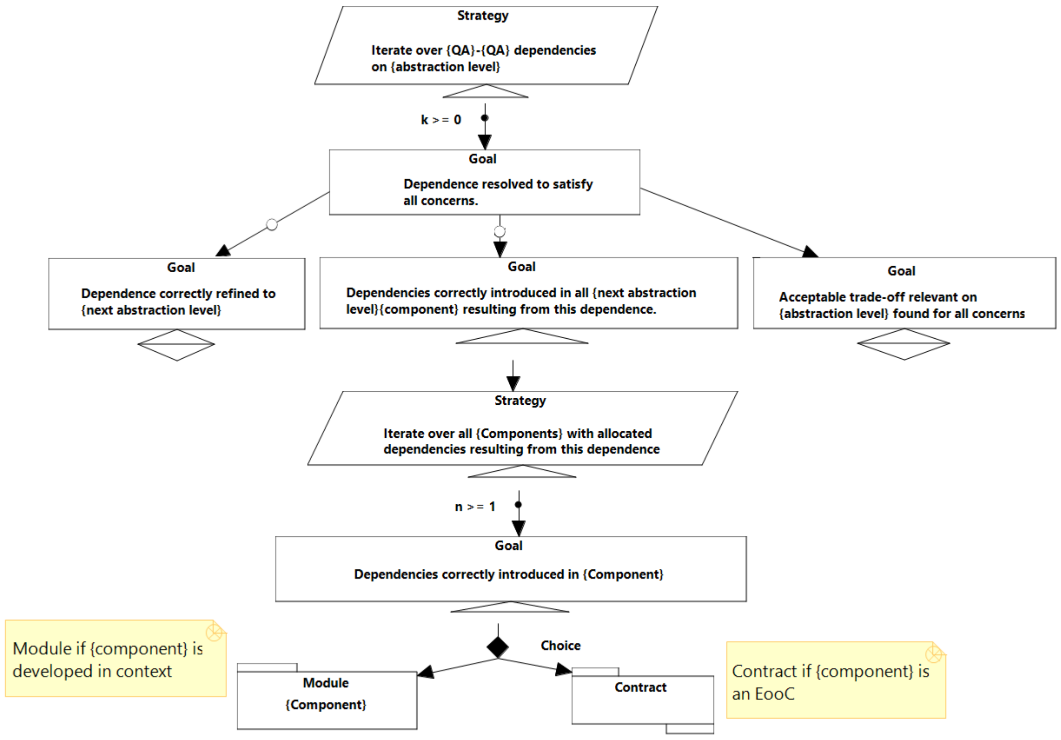
Strategy
Iterate over {QA}-{QA} dependencies on {abstraction level}
k >= 0
Goal
Dependence resolved to satisfy all concerns.
Goal
Dependence correctly refined to {next abstraction level}
Goal
Dependencies correctly introduced in all {next abstraction level}{component} resulting from this dependence.
Goal
Acceptable trade-off relevant on {abstraction level} found for all concerns
Strategy
Iterate over all {Components} with allocated dependencies resulting from this dependence
n >= 1
Goal
Dependencies correctly introduced in {Component}
Module if {component} is developed in context
Choice
Module
{Component}
Contract
Contract if {component} is an EooC

Figure 9 Pattern for an interplay refinement.

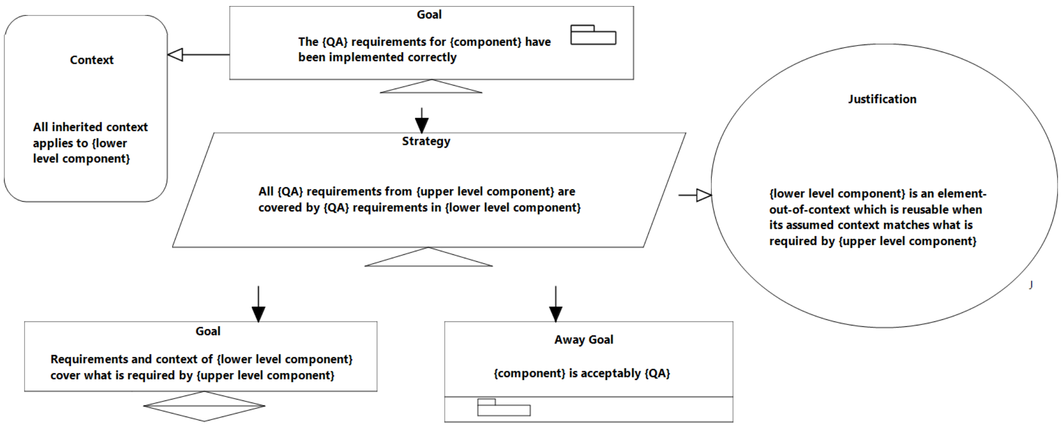
Goal
The {QA} requirements for {component} have been implemented correctly
Context
All inherited context applies to {lower level component}
Strategy
All {QA} requirements from {upper level component} are covered by {QA} requirements in {lower level component}
Justification
{lower level component} is an element-out-of-context which is reusable when its assumed context matches what is required by {upper level component}
J
Goal
Requirements and context of {lower level component} cover what is required by {upper level component}
Away Goal
{component} is acceptably {QA}

Figure 10 Pattern for a contract between system/component and an element-out-of-context.

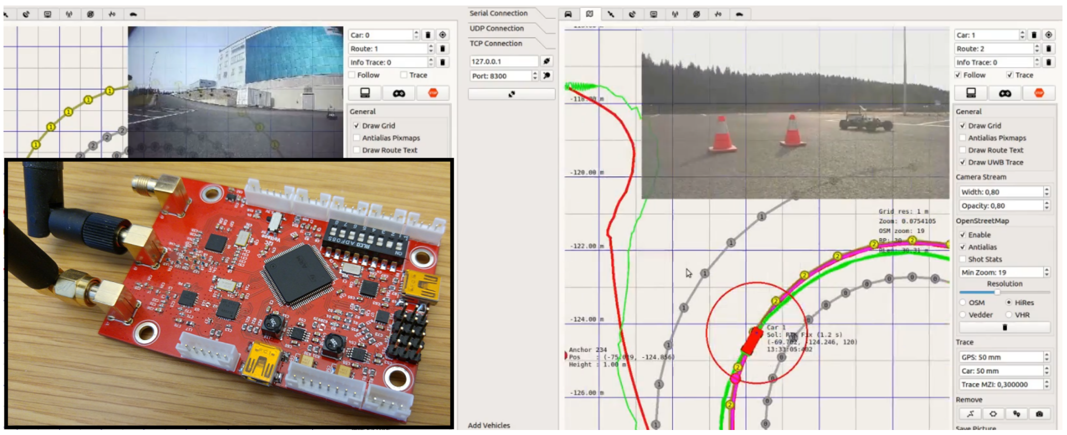

Figure 11 Demonstrator with model cars using the PE (inset) for navigation.

ISO 26262 is used as safety standard and a working draft of ISO/SAE 21434[2] for cybersecurity. Fig. 11 (inset) shows the hardware for PE containing a satellite navigation receiver which is used in conjunction with correction data for enhanced precision. To complete the use case, PE is matched to the hypothetical context of an ADS feature - highway autopilot, where it is used to provide accurate absolute (i.e. on a map) position. Fig. 11 also shows a demonstrator environment for this feature with autonomous model cars.

A detailed description of the function is beyond the scope of this paper; however, it has functional requirements which are analyzed for safety and security risks according to both standards, resulting in safety goals (top-level safety requirements) and security goals. A simplified version of one functional requirements is: *The automated driving mode may only be activated on roads certified for ADS vehicles*. The ISO 26262 HA&RA results in safety goals for the ADC. A safety goal related to the stated requirement is: *ADC may only be activated on certified roads*[3]. Since the function is only designed to work within the parameters given in the functional requirement, its behavior is undefined if enabled anywhere else, thus resulting in high risk of harm. For cybersecurity, a threat analysis and risk assessment (TA&RA) is used to elicit security goals. A security goal with a dependency to the mentioned safety goal is: *Integrity protection against spoofing to fulfil ADC may only be activated on certified roads.*

For space reasons the entire argument for the case study cannot be shown. However, Fig. 12 shows some parts of interest: (a) dependence between the safety and security goals discussed above; (b) the same dependence refined to functional level, (c) example of where requirements from ISO 26262 have been connected to the assurance case (HA&RA forms a tree of its own ending in requirements from the standard, this tree has been automatically generated to a module), and (d) reference to the contract between function and positioning EooC.

5 Conclusions

In this paper, our goal was to propose a structured way to build the argument for a multi-concern assurance case of a complex dependability-critical system such as a CAD function, and demonstrate its feasibility by an example. Our claim is that the complexity can be

[2] A coming cybersecurity standard for the automotive domain.
[3] The actual safety and security goals also have integrity levels but as they are not relevant for the example we have omitted them.

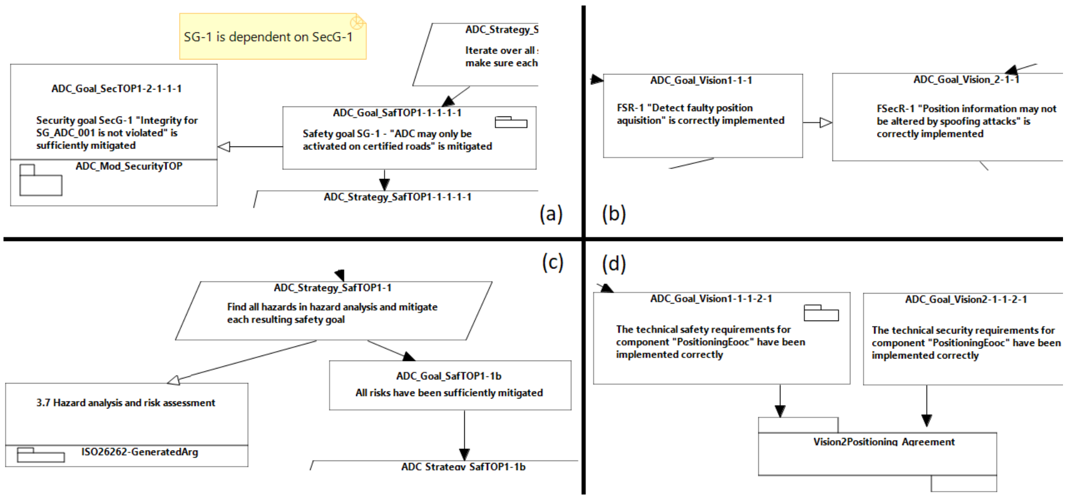
SG-1 is dependent on SecG-1
ADC_Strategy_S
Iterate over all
make sure each
ADC_Goal_SecTOP1-2-1-1-1
Security goal SecG-1 "Integrity for SG_ADC_001 is not violated" is sufficiently mitigated
ADC_Mod_SecurityTOP
ADC_Goal_SafTOP1-1-1-1-1
Safety goal SG-1 - "ADC may only be activated on certified roads" is mitigated
ADC_Strategy_SafTOP1-1-1-1-1
(a)
(b)
ADC_Goal_Vision1-1-1
FSR-1 "Detect faulty position aquisition" is correctly implemented
ADC_Goal_Vision_2-1-1
FSecR-1 "Position information may not be altered by spoofing attacks" is correctly implemented
(c)
ADC_Strategy_SafTOP1-1
Find all hazards in hazard analysis and mitigate each resulting safety goal
3.7 Hazard analysis and risk assessment
ISO26262-GeneratedArg
ADC_Goal_SafTOP1-1b
All risks have been sufficiently mitigated
ADC Strategy SafTOP1-1b
(d)
ADC_Goal_Vision1-1-1-2-1
The technical safety requirements for component "PositioningEooc" have been implemented correctly
ADC_Goal_Vision2-1-1-2-1
The technical security requirements for component "PositioningEooc" have been implemented correctly
Vision2Positioning Agreement

Figure 12 Snippets from argument for case study.

managed by a traceable argument structure, with an attached rationale to every branching in order to keep track of the reasoning behind the design. With this structure the overall design can also be aggregated and contain re-usable components. The structure follows each concern, with dependencies at certain interaction points. The interaction can be predicted because the argument structure also reflects the development lifecycles of the concerns. When the interaction between the concerns is planned and limited, as we propose, there is a good possibility too keep the benefits from co-engineering, without the extra effort of high frequency interaction between different disciplines such as safety and security.

It should be noted that even if a structured approach makes it easier to manage large complex systems, the approach would still require good tool support to be feasible as the arguments can become very large. Traceability and compliance management, management of argument modules, and automation of argument integrity checks are examples where tools are helpful. There is ample opportunity for automation, for instance detecting nodes that have not been developed or instantiated, or solution nodes with no references to actual evidence. Combination with semi-formal notations for goals/requirements to allow for even better control of structure and more checks for possible omissions is yet another possibility to increase automation opportunities. Some tools such as OpenCert already contain many of these features. Another issue we have not discussed in the paper is how to include assurance in the actual development workflow. For instance, today many organizations are adopting agile practices to allow for more frequent product updates. This is an issue we are currently exploring in our continued work.

References

1 AMASS deliverable D4.3: Design of the AMASS tools and methods for multiconcern assurance, 2018. [Accessed 17-April-2019]. URL: `https://www.amass-ecsel.eu/`.

2 AMASS deliverable D4.8: Methodological guide for multiconcern assurance(b), 2018. [Accessed 17-April-2019]. URL: `https://www.amass-ecsel.eu/`.

3 John Birch, Roger Rivett, Ibrahim Habli, Ben Bradshaw, John Botham, Dave Higham, Peter Jesty, Helen Monkhouse, and Robert Palin. Safety cases and their role in ISO 26262 functional safety assessment. In *32nd International Conference on Computer Safety, Reliability, and Security, SAFECOMP*, pages 154–165. Springer, 2013.

4 John Birch, Roger Rivett, Ibrahim Habli, Ben Bradshaw, John Botham, Dave Higham, Helen Monkhouse, and Robert Palin. A Layered Model for Structuring Automotive Safety Arguments (Short Paper). In *10th European Dependable Computing Conference, EDCC*, pages 178–181. IEEE, 2014.

5 Thomas Chowdhury, Chung-Wei Lin, BaekGyu Kim, Mark Lawford, Shinichi Shiraishi, and Alan Wassyng. Principles for systematic development of an assurance case template from ISO 26262. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW*, pages 69–72. IEEE, 2017.

6 Georgios Despotou and Tim Kelly. An Argument-Based Approach for Assessing Design Alternatives and Facilitating Trade-offs in Critical Systems. *Journal of System Safety*, 43(2):22, 2007.

7 Ashlie B Hocking, John Knight, M Anthony Aiello, and Shinichi Shiraishi. Arguing software compliance with ISO 26262. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW*, pages 226–231. IEEE, 2014.

8 ISO. ISO 26262:2018 Road vehicles – Functional safety, 2018.

9 ISO/IEC. ISO/IEC 15026-2:2011 Systems and software engineering – Systems and software assurance – Part 2: Assurance case, 2011.

10 ISO/IEC. ISO/IEC 15026-2:2015 Systems and software engineering – Systems and software assurance – Part 3: System integrity levels, 2015.

11 Nikita Johnson and Tim Kelly. An Assurance Framework for Independent Co-assurance of Safety and Security. In *36th International System Safety Conference, ISSC*, 2018.

12 Helmut Martin, Robert Bramberger, Christoph Schmittner, Zhendong Ma, Thomas Gruber, Alejandra Ruiz, and Georg Macher. Safety and security co-engineering and argumentation framework. In *International Conference on Computer Safety, Reliability, and Security*, pages 286–297. Springer, 2017.

13 Helmut Martin, Martin Krammer, Robert Bramberger, and Eric Armengaud. Process- and product-based lines of argument for automotive safety cases. In *ACM/IEEE 7th International Conference on Cyber-Physical Systems, ICCPS*, 2016.

14 OpecCert contributors. OpenCert. [Accessed 17-April-2019]. URL: `https://www.polarsys.org/projects/polarsys.opencert`.

15 Rob Palin, David Ward, Ibrahim Habli, and Roger Rivett. ISO 26262 safety cases: Compliance and assurance. In *6th IET International Conference on System Safety*. IET, 2011.

16 Robert Palin and Ibrahim Habli. Assurance of Automotive Safety–A Safety Case Approach. In *29th International Conference on Computer Safety, Reliability, and Security, SAFECOMP*, pages 14–17. Springer, 2010.

17 SAE. SAE J3016:201806 - SURFACE VEHICLE RECOMMENDED PRACTICE - Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles, 2018.

18 SCSC Assurance Case Working Group Contributors. GSN Community Standard Version 2, 2018. [Accessed 17-April-2019]. URL: `https://scsc.uk/r141B:1?t=1`.

19 Kenji Taguchi, Daisuke Souma, and Hideaki Nishihara. Safe & sec case patterns. In *33rd International Conference on Computer Safety, Reliability, and Security, SAFECOMP*, pages 27–37. Springer, 2014.

Sustainable Security & Safety: Challenges and Opportunities

Andrew Paverd
Microsoft Research Cambridge, UK
andrew.paverd@ieee.org

Marcus Völp[1]
University of Luxembourg, Luxembourg
marcus.voelp@uni.lu

Ferdinand Brasser[1]
TU Darmstadt, Germany
ferdinand.brasser@trust.tu-darmstadt.de

Matthias Schunter[1]
Intel Labs, Darmstadt, Germany
matthias.schunter@intel.com

N. Asokan
Aalto University, Finland
asokan@acm.org

Ahmad-Reza Sadeghi[1]
TU Darmstadt, Germany
ahmad.sadeghi@trust.tu-darmstadt.de

Paulo Esteves-Veríssimo
University of Luxembourg
paulo.verissimo@uni.lu

Andreas Steininger
TU Wien, Austria
steininger@ecs.tuwien.ac.at

Thorsten Holz
Ruhr-University Bochum, Germany
thorsten.holz@rub.de

Abstract

A significant proportion of today's information and communication technology (ICT) systems are entrusted with high value assets, and our modern society has become increasingly dependent on these systems operating safely and securely over their anticipated lifetimes. However, we observe a mismatch between the lifetimes expected from ICT-supported systems (such as autonomous cars) and the duration for which these systems are able to remain safe and secure, given the spectrum of threats they face. Whereas most systems today are constructed within the constraints of foreseeable technology advancements, we argue that long term, i.e., *sustainable security & safety*, requires anticipating the unforeseeable and preparing systems for threats not known today. In this paper, we set out our vision for sustainable security & safety. We summarize the main challenges in realizing this desideratum in real-world systems, and we identify several design principles that could address these challenges and serve as building blocks for achieving this vision.

2012 ACM Subject Classification Security and privacy → Systems security; Software and its engineering → Software reliability

Keywords and phrases sustainability, security, safety

Digital Object Identifier 10.4230/OASIcs.CERTS.2019.4

Related Version A full version of the paper is available at [14], `http://www.icri-cars.org/wp-content/uploads/2019/01/s3-vision.pdf`

Acknowledgements This work was supported by the Intel Research Institute for Collaborative Autonomous and Resilient Systems (ICRI-CARS). The authors thank Muhammad Shafique for his helpful suggestions on this manuscript.

[1] corresponding author

1 Introduction

With the exception of a handful of systems, such as the two Voyager spacecraft control systems and the computers in the US intercontinental ballistic missile silos,[1] information and communication technology systems (ICT) rarely reach a commercially viable lifetime that ranges into the 25+ years we have come to expect from long-lived systems like cars[2], as shown in Figure 1. Worse, rarely any networked ICT system stays secure over such a lifetime, even if actively maintained.

Despite this potential risk, current ICT subsystems are already being integrated into systems with significantly longer design lifetimes. For example, electronic control units in modern (self-driving) cars, the networked control systems in critical infrastructure (e.g., cyber-physical systems in power plants and water treatment facilities), and computer controlled building facilities (a.k.a. smart buildings) are all examples of this mismatch in design lifetimes. This is also applicable beyond cyber-physical systems: for example, genomic data is privacy-sensitive for at least the lifetime of the person (if not longer), and yet it is being protected by cryptographic algorithms with a significantly shorter expected lifespan [6]. For the currently envisioned limited lifetime, the (functional) safety community has developed techniques to prevent harm despite accidental faults [9]. However, for the ICT systems in the above scenarios we aim to preserve security and safetime for the design lifetime, and often beyond. This means that they need to preserve the value and trust in the assets entrusted to them, their safety properties (i.e., resilience to accidental faults) *and* their security properties (i.e., resilience to malicious faults, such as targeted and persistent attacks). Even when compared to existing approaches to achieve resilience (such as the safety approaches mentioned above), these lifetimes translate into ultra-long periods of secure and safe operation.

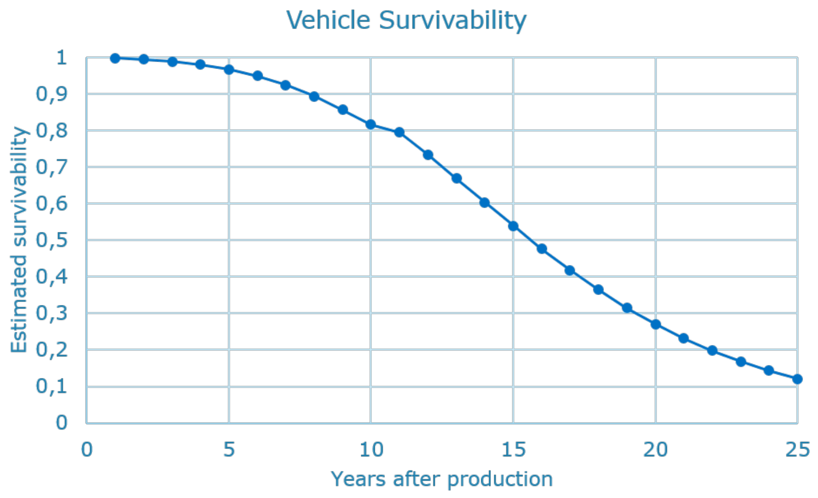

Figure 1 Survival probability of passenger cars by age [2] (based on 1977-2003 NVPP data).

To address this impending challenge, we introduce a new paradigm, which we call *sustainable security & safety* (S3). The central goal is that a system should be able to maintain its security and safety, but desirably also its functionality for at least its design

[1] `http://www.dailymail.co.uk/news/article-2614323/Americas-feared-nuclear-missile-facilities-controlled-computers-1960s-floppy-disks.html`
[2] `https://www.aarp.org/money/budgeting-saving/info-05-2009/cars_that_last.html`

lifetime. This is particularly relevant for systems that have significantly longer design lifetimes than their ICT subsystems (e.g., modern cars, airplanes, and other cyber physical systems). In its broadest sense, S3 encompasses both technical and non-technical aspects, and some of the arising challenges span both aspects.

From a technical perspective, S3 brings together aspects of two well-studied technical fields: *security* and *dependability*. Security research typically begins by anticipating a certain class of adversary, characterized by a threat model based on well known pre-existing threats at the time the system is designed. From this model, defense mechanisms are then derived for protecting the system against the anticipated adversaries or for recovering from these known attacks. Dependability, on the other hand, begins with the premise that some components of a system will fail, and investigates how to tolerate faults originating from these known subsystem failures. Again, the type, likelihood, and consequences of faults are assumed to be known and captured in the fault and fault-propagation models.

The term *security-dependability gap* [10] is a prototypical example of why established solutions from the dependability field cannot be directly used to address security hazards, and vice-versa. This arises from the way risks are assessed in safety-critical systems: safety certifications assess risks as a combination of stochastic events, whereas security risks arise as a consequence of intentional malicious adversaries, and thus cannot be accurately characterized in the same way. What is a residual uncertainty in the former, becomes a strong likelihood in the latter.

Therefore, there are two defining characteristics of S3:

- Firstly, it aims to *bridge the safety-security gap* by considering the complementarity of these two fields as well as the interplay between them, and addressing the above-mentioned problems under a common body of knowledge, seeking to prevent, detect, remove and/or tolerate both accidental faults and vulnerabilities, and malicious attacks and intrusions.
- Secondly, it aims to *protect systems beyond the foreseeable horizon* of technological (and non-technological) advances and therefore cannot be based solely on the characterizations of adversaries and faults as we know them today. Instead we begin from the premise that a long-lived system will face changes, attacks, and accidental faults that were not possible to anticipate during the design of the system.

The central challenge is therefore to design systems that can maintain their security and safety properties in light of these unknowns.

2 System Model

Sustainable security & safety is a desirable property of any critical system, but is particularly relevant to long-lived systems, especially those that are easily accessible to attackers. These systems are likely to be comparatively complex, consisting of multiple subsystems and depending on various external systems. Without loss of generality, we use the following terminology in this paper, which is aligned with other proposed taxonomies, such as [3]:

- **System:** a composition of multiple subsystems. The subsystems can be homogeneous (e.g., nodes in a swarm) or heterogeneous (e.g., components in an autonomous vehicle).
- **Failure:** degradation of functionality and/or performance beyond a specified threshold.
- **Error:** deviation from the correct service state of a system. Errors may lead to subsequent failures.
- **Fault:** adjudged or hypothesized cause of an error (e.g., a dormant vulnerability of the system through which adversaries may infiltrate the system, causing an error which leads to a system failure).

- **System failure:** failure of the overall system (typically with catastrophic effect).
- **Subsystem failure:** failure of an individual subsystem. The overall system may be equipped with precautions to tolerate certain subsystem failures. In this case, subsystem failures can be considered as faults in the overall system.
- **Single point of failure:** any subsystem whose failure alone results in system failure.

S3 aims to achieve the following two goals:

1. **Avoid system failure:** The primary goal is to avoid failure of the overall system, including those originating from unforeseeable causes and future attacks. Note that avoiding system failure refers to both the safety and security properties of the system (e.g., a breach of data confidentiality or integrity could be a system failure, for certain types of systems).
2. **Minimize overheads and costs:** Anticipating and mitigating additional threats generally increases the overheads (e.g., performance and energy) as well as the initial costs (e.g., direct costs and increased design time) of the system.

In addition, higher system complexity (e.g., larger code size) may lead to increased fault rates and an increased attack surface. On the other hand, premature system failure may also have associated costs, such as downtime, maintenance costs, or penalties. Therefore, a secondary goal of S3 is to develop technologies and approaches that minimize all these potential overheads and costs.

3 Challenges of Long-Term Operation

In our S3 paradigm, we accept that we cannot know the precise nature of the attacks, faults, and changes that a long-lived system will face during its lifetime. However, we can identify and reason about the fundamental classes of events that could arise during the system's lifetime and that are relevant to the system's security and dependability. Although these events are not exclusive to long-term operation, they become more likely as the designed lifetime of the system increases. We first summarize the main classes of challenges, and then discuss each in detail in the following subsections.

Subsystem failures

In consequence of the above uncertainty about the root cause and nature of faults leading to subsystem failure, traditional solutions, such as enumerating all possible faults and devising mitigation strategies for each fault class individually, are no longer applicable. Instead, reasoning about these causes must anticipate a residual risk of unknown faults ultimately leading to subsystem failures and, treating them as faults of the system as a whole, mechanisms included that are capable of mitigating the effects of such failures at the system level. Subsystem failures could be the result of random events or deliberate attacks, possibly due to new types of attack vectors being discovered. In the most general case, any type of subsystem could fail, including hardware failures, software attacks, and failures due to incorrect specifications of these subsystems. Subsystem failures may be hardware failures, software failures, subsystem compromises or specification failures. The open issues in these areas include:

Hardware failures. How can we protect spares (set our to take over in case of successful attacks or persistent faults) from environmental and adversarial influences, such that they remain available when they are required? How can we assert the sustainability of emerging material circuits, without at the same time giving adversaries the tools to stress and ultimatlely break these circuits? How can we protect confidential information in subsystems? How can we prevent one compromised hardware subsystem from compromising the integrity of others? How can we prevent adversaries from exploiting safety/security functionality of excluded components? How can we model erroneous hardware behavior? And, how can we construct inexpensive, fine grain isolation domains to confine such errors?

Software failures. How to design systems that can detect and isolate software subsystem failures? How to transfer software attack mitigation strategies between domains (e.g., PC to embedded)?

Subsystem compromise. How to recover a system when *multiple* subsystems are compromised? How to detect subsystem compromised by a stealthy adversary? How to react to the detection of a (potentially) compromised subsystem? How to prevent the leakage of sensitive information from a compromised subsystem? How to securely re-provision a subsystem after all secrets have been leaked?

Specification failures. How to design subsystems that may fail at the implementation, but not at the specification level (and at what costs)? If specification faults are inevitable, how to design systems in which subsystems can follow different specifications whilst providing the same functionality, in order to benefit from diversity specifications and assumptions? How to recover when one of the essential systems has been broken due to a specification error (e.g., how to recover from a compromised cryptographic subsystem)?

Requirement changes

The requirements of the system could change during its lifetime. For example, the security requirements of a system could change due to new regulations (e.g., the EU General Data Protection Regulation) or shifts in societal expectations. The expected lifetime of the system itself could even be changed subsequent to its design. Requirement changes may be due to:

Regulatory changes. How to retroactively change the designed security, privacy, and/or safety guarantees of a system? How to prove that an already-deployed system complies with new regulations?

User expectation changes. How can a system be extended and adapted to meet new expectations after deployment? How to demonstrate to users and other stakeholders that an already-deployed system meets their new expectations?

Design lifetime changes. How to determine whether a deployed system will retain its safety and security guarantees for an even longer lifetime? How to further extend the safety and security guarantees of a deployed system?

Environmental changes

The environment in which the system operates could change during the system's design lifetime. This could include changes in other interdependent systems. For example, the United States government can selectively deny access to the GPS system, which is used by autonomous vehicles for localization.

New threat vectors. How to tolerate failure of subsystems due to unforeseeable threats? How to avoid single points of failure that could be susceptible to unforeseen threats? How to improve the modeling of couplings and dependencies between subsystems such that the space of "unforeseeable" threats can be minimized?

Unilateral system/service changes. How to design systems such that any third-party services on which they depend can be safely and securely changed? How can a system handle unilateral changes of (external) systems or services?

Third-party system/service fails. How to design systems such that any third-party services on which they depend can be safely and securely changed *after they have already failed*? How can a system handle the failure or unavailability of external services?

Maintenance resource becomes unavailable. How to identify all maintenance resources required by a system? How to maximize the *maintenance lifetime* of a system whilst minimizing cost? How to continue maintaining a system when a required resource becomes completely unavailable? How to ensure that a system remains secure and safe even under a new maintainer?

Maintainer changes

Most systems have the notion of a maintainer – the physical or logical entity that maintains the system for some part of its design lifetime. For example, in addition to the mechanical maintenance (replacing brakes, oil, etc.) Tesla vehicles regularly receive over-the-air software updates from the manufacturer.[3] For many systems, the maintainer needs to be trusted to sustain the security and safety of the maintained system. However, especially in long-lived systems, the maintainer may change (e.g., the vehicle is instead maintained by a third party service provider), which gives rise to both technical and non-technical challenges. Maintenance change requires answering:

Implementing a change of maintainer. How to securely inform the system that a change of maintainer has taken place?

System becomes unmaintained. How can a system decide that it is no longer maintained? How should an unmaintained system behave?

Ownership changes

Finally, if a system has the notion of an owner, it is possible that this owner will change over the lifetime of the system, especially if the system is long-lived. For example, vehicles are often resold, perhaps even multiple times, during their design lifetimes. Change of ownership has consequences because the owner usually has privileged access to the system. A system may also be collectively owned (e.g., an apartment block may be collectively owned by the owners of the individual apartments). Ownership may be:

[3] `https://www.tesla.com/support/software-updates`

Cooperative. How to securely inform the system about the change in ownership, without opening a potential attack vector? How to erase sensitive data during transfer of ownership, without allowing the previous owner to later erase usage/data of the new owner?

Non-cooperative. How to automatically detect non-cooperative ownership change? How to erase sensitive data after loss of ownership, without allowing the previous owner to erase usage/data of the new owner?

4 Technical Implications

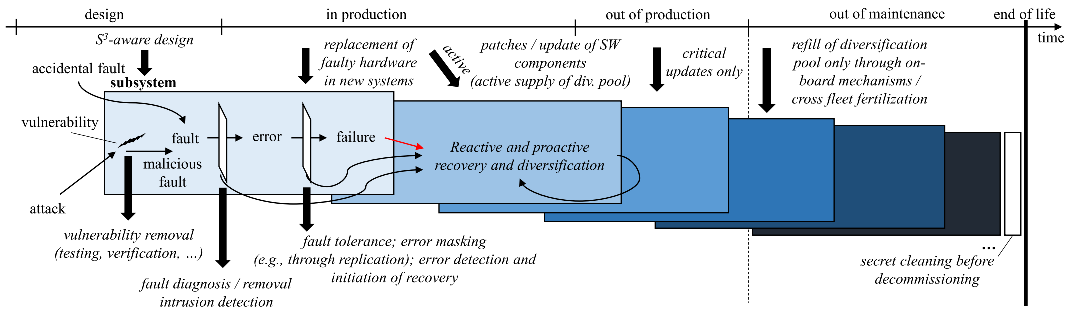
design
in production
out of production
out of maintenance
end of life
time
S³-aware design
accidental fault
subsystem
vulnerability
fault
malicious fault
attack
error
failure
replacement of faulty hardware in new systems
active
patches / update of SW components (active supply of div. pool)
critical updates only
refill of diversification pool only through on-board mechanisms / cross fleet fertilization
Reactive and proactive recovery and diversification
vulnerability removal (testing, verification, ...)
fault diagnosis / removal intrusion detection
fault tolerance; error masking (e.g., through replication); error detection and initiation of recovery
...
secret cleaning before decommissioning

Figure 2 Means to attain dependability and security.

The challenges identified in Section 3 lead to a number of technical implications, which we highlight in the following before presenting more concrete proposals to address sustainable security & safety in the next section. In particular, the realization that any subsystem may fail, especially given currently unforeseeable threat vectors, and the realization that subsystem failure is a fault in the overall system, demands a principled treatment of faults.

Although safety and security independently developed solutions to characterize the risk associated with faults, we observe a gap to be closed, not only for automotive systems [10], but also in any ICT system where risks are assessed based on the probability of occurrence. Once exposed to adversaries, it is no longer sufficient to capture the likelihood of natural causes coming together as probability distributions. Instead, probabilistic risk assessment must take into consideration the incentives, luck, and intents of increasingly well-equipped and highly skilled hacking teams. The conclusion may well be high risk probabilities, in particular when irrational hackers (such as cyberterrorists) are taken into account. Also, naively applying techniques, principles, and paradigms used in isolation in either safety or security will not solve the whole problem, and worse, may interfere with each other. Moreover, to reach sustainability, any such technique must withstand long-term operation, significantly exceeding that of recent approaches to resilience.

For example, executing a service in a triple modular redundant fashion increases tolerance against accidental faults in one of the replicas. However, it does not help to protect a secret passed to the individual replicas by encrypting this secret for each replica and with the replica's key. Once the key of a single compromised replica is extracted, confidentiality is breached and the secret revealed. Further, if a replica fails due to an attack rather than a random fault, replaying the attack after resetting this replica will often recreate this failure and eventually exhaust the healthy majority.

To approach sustainable security & safety, we need a common descriptor for the root-cause of problems in both fields – *faults* – and a principled approach to address them [21, 20]. As already pointed out in [3], faults can be accidental, for example, due to natural causes, following a stochastic distribution and hence justifying the argument brought forward in safety cases that the residual risk of a combination of rare events is within a threshold of accepted risks. However, faults can also be malicious, caused by intentional adversaries exploiting reachable vulnerabilities to penetrate the system and then having an up to 100% chance of triggering the combination of rare events that may lead to catastrophe. Faults can assume several facets, and the properties affected are the union of those both fields are concerned with: reliability, availability, maintainability, integrity, confidentiality, etc.

Returning to the above example, treating a confidentiality breach as a fault, it becomes evident why encrypting the whole secret for each replica individually constitutes a single point of failure. It also gives rise to possible solutions, such as secret sharing [16], but further research is required to ensure long-term secrecy for those application fields that require this, e.g., simple secret sharing schemes do not tolerate a mobile adversary that subsequently compromises and releases one replica at a time.

In the long run, failure of individual subsystems within the lifetime of the overall system becomes an expected occurrence. We therefore distinguish normal failure (e.g., ageing causing unavailability of a subsystem's functionality) from catastrophic failure (causing unintended system behavior), and devise means to attain dependability and security despite both.

4.1 S3 Lifecycle

Figure 2 sketches the path towards attaining dependability and security, and principled methods that can be applied over the lifetime of the system to obtain sustainable security & safety. Faults in subsystems manifest from attacks exploiting reachable vulnerabilities in the specification and implementation of the subsystem or from accidental causes. Without further provisioning, faults may manifest in errors, which may ultimately lead to failure of the subsystem. Here we take only the view of a single subsystem, which may well fail without inevitably implying system failure. More precisely, when extending the picture in Figure 2 to the overall system, subsystem failures manifest as system faults, which must be caught at the latest at the fault tolerance step. It may well be too late to recover the system as a whole after a failure has manifested, since this failure may already have compromised security or caused a catastrophe (indicated by the red arrow).

4.1.1 Design

Before deployment, the most important steps towards sustainable security & safety involve preparing the system as a whole for the consequences of long-term operation. In the next section, we sketch early insights what such a design may involve. Equally important is to reduce the number of vulnerabilities in the system, to raise the bar for adversaries. Fault and vulnerability prevention starts from a rigorous design and implementation, supported for example by advanced testing techniques such as fuzzing [12, 13, 7, 8]. However, we also acknowledge the limitations of these approaches and consequently the infeasibility of zero-defect subsystems once they exceed a certain complexity, in particular in the light of unforeseeable threats. For this reason, intrusion detection and fault diagnosis and fault and vulnerability removal starts after the system goes into production,[4] which also is imperfect

[4] We consider design-time penetration testing as part of the vulnerability removal process.

in detecting only a subset of intrusions and rarely those following unknown patterns while remaining within the operational perimeter of the subsystem. It is important to notice that despite the principled imperfection of fault, vulnerability and attack prevention, detection and removal techniques, they play an important role in increasing the time adversaries need to infiltrate the system and in assessing the current threat level, the system is exposed to.

4.1.2 In Production

While in production, replacement of faulty hardware components is still possible, by providing replacement parts to those systems that are already shipped and by not shipping new systems with known faulty parts. Software updates remain possible during the whole time when the system remains under maintenance, although development teams may already have been relocated to different projects after the production phase.

Fault tolerance, that is, a subsystem's ability to gracefully reduce its functionality to a non-harmful subset (e.g., by crashing if replicas start to deviate) or to mask errors (e.g., by outvoting the behaviour of compromised or malicious replicas behind a majority of healthy replicas, operating in consensus) forms the last line of defense before the subsystem failure manifests as a fault. The essential assumption for replication to catch errors not already captured by the fault and error removal stages is fault-independence of all replicas, which is also known as absence of common mode faults. Undetected common mode faults bear the risks of allowing compromise of all replicas simultaneously, which gives adversaries the ability to overpower the fault tolerance mechanisms. Crucial building blocks for replication-based fault tolerance are therefore the rejuvenation of replicas to repair already compromised replicas and in turn maintain over time the required threshold of healthy replicas (at least one for detecting and crashing and at least $f+1$ to mask up to f faults) and diversification to avoid replicas from failing simultaneously and to cancel adversarial knowledge how replicas can be compromised.

4.1.3 Out of Production

As long as the system is maintained, the rejuvenation and diversification infrastructure for the system may rely on an external supply of patches, updates and new, sufficiently diverse variants for the pool of diverse subsystem images.

4.1.4 Out of Maintenance

Once the system falls out of active maintenance, replenishment of this pool is limited to on-board diversification mechanisms (such as binary obfuscation[5]) or through fleet-wide cross-fertilization, by exchanging diagnosis data between systems of the same kind, which allows one system to learn from the intrusions that happened to its peers (e.g., by mimicking gene crosscopying in patch generation for defense). The final state before the end of life of the system is a reliable cleaning step of all secrets that remain until this time, followed by a graceful shutdown of the system.

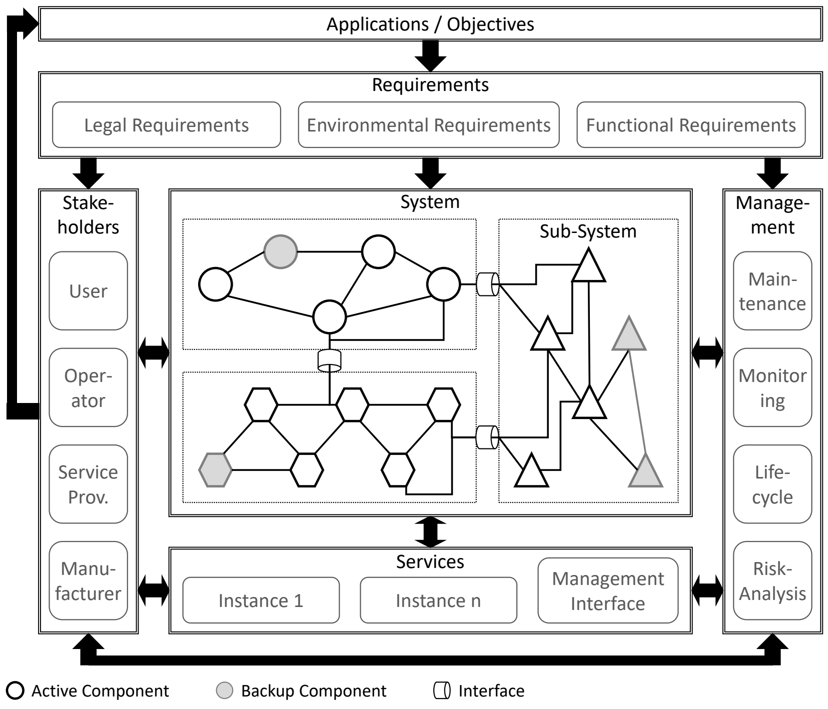

Figure 3 Abstract view of a sustainable system and its surroundings.

5 Design Principles

Figure 3 sketches one potential architecture for sustainable security & safety and shows the abstract view of a sustainable systems and its connections/relations with its surroundings. A system interacts (possibly in different phases of its life cycle) with different stakeholders. The stakeholders (or a subset thereof) usually define the applications and objectives of a systems, e.g., the manufacturer and developer define the primary/intended functionality of a systems. The system's intended applications or objectives, as well as external factors such legal frameworks, define the overall requirements a system must fulfill.

The center of Figure 3 shows the overall system that can be composed from multiple subsystems, each of which is a combination of multiple components. Components of a subsystem are connected with one another. Backup components (marked in grey) are required to achieve fault tolerance, i.e., if one component of a subsystem fails it can be (automatically) replaced by a backup component.[6]

Subsystems are connected and interact via defined interfaces (shown as tubes). As long as the interfaces and the individual subsystems are fault tolerant, the overall system can inherit this property.[7]

Sustainable systems usually do not operate in isolation. They often rely on external services, e.g., cloud services to execute computation intensive tasks. If these services are critical for the operation of the system, the resiliency of these services is relevant, e.g.,

[5] `https://www.defcon.org/images/defcon-17/dc-17-presentations/defcon-17-sean_taylor-binary_obfuscation.pdf`
[6] Backup components can also be online/active to reduce the load per component as long as not failure has occurred.
[7] If an individual subsystem cannot provide the required fault tolerance level, the overall system requires redundancy with regard to that subsystem.

multiple instances of the service must be available, and the connection to the external service must be reliable, as well.

The system's management includes technical management as well as abstractly controlling the system and its operation. This has to cover both the management of the system itself and the external services upon which the system relies. The management can be performed by a single entity (stakeholder) or distributed among different stakeholders, e.g., the device manufacturer provides software updates for the system while the user configures and monitors it.

Based on the challenges discussed in the preceding section, we can already deduce certain architectural requirements and design principles for achieving S3:

- **Well-defined Components and Isolation**, limiting interaction to well-defined and constrained interfaces, to ensure complexity can be handled and to confine faults to causing components.
- **Avoid Single Points of Failure.** For systems in long-term operation, it must be expected that any subsystem could fail, especially when exposed to unforeseen or unforeseeable attacks or changes. Anticipating this possibility, it is clear that no single subsystem can be responsible for the safety and security of the system as a whole.
- **Multiple Lines of Defense.** Individual defenses can be overcome by adversaries, hence, relying on a single defense mechanism represents a single point of failure. Multiple lines of defense are needed to protect the system. Control-flow integrity (CFI) [1], for instance, has been attacked by full-function reuse attacks [5, 15] while randomization approaches suffer from information leakage [17]. Combining both can increase the security, e.g., by relying on randomization to mitigate the above attack to CFI.
- **Long-term Secrets and Confidentiality.** Ensuring the confidentiality of data over long periods of time is very challenging since data, once leaked, cannot become confidential again. To prevent the leakage of secret information their use should be minimized. Additionally, single subsystem should not be allowed to access a secret as a whole, as this subsystem would become a single point of failure with respect to the secret information's confidentiality.

  Let us give an example, detailled in [22]. Combining secret sharing, permanently re-encrypted data silos, function shipping into trusted execution environments, and sanitizing before revealing, it is possible to give access, even to bulk data, when it is required, but only in the form necessary. Rarely, applications require access to raw data. Instead, most just depend on aggregate, derived information which reveals much less of the secrets used. Revealing the classification of a deep neural network for a picture, while protecting the weights [19] is an example of such a secret protection scheme.
- **Robust Input Handling.** (Sub)systems should not make assumptions with respect to the inputs it expects. This holds true for external as well as internal inputs. A robust system should incorporate: (a) it should cope well with noisy and uncertain real-world inputs, and (b) express its own knowledge and uncertainty of a proposed output despite unforeseen input [11].
- **Contain Subsystem Failure.** An immediate conclusion from the requirement to tolerate arbitrary failures is the need to confine each subsystem into an execution environment that acts as fault containment domain and in which the subsystem may be rejuvenated to re-establish the functionality it provides.
- **Replicate to Tolerate Subsystem Failure.** If the functionality provided by the failing subsystem cannot be compensated by lower-level components (possibly at a different quality of service), the subsystem must be replicated to mask failure of individual replicas behind a majority of healthy replicas operating in consensus.

- **Diversify Nodes and Components.** Replication is not sufficient to evade attacks [4]; diversification is needed to avoid common mode faults and cancel adversary knowledge [18].
- **Adaption and reconfiguration** is needed to stay ahead of adversaries, but also to adjust to environmental changes.
- **Minimize Assumptions.** Attackers may find new ways of violating assumptions to defeat safety or security measures that rely on them. Minimizing assumptions to those substantially required for the implementation, mitigates this threat.
- **Simplicity and Verifiability.** While due to performance demands it is out of reach to keep the whole system simple, the trusted core typically required in fault-tolerant and secure architectures can be kept sufficiently small. Such a small and simple unit is furthermore less likely to suffer from unforeseen threats.

6 Conclusion

Achieving sustainable security & safety in real-world systems is a non-trivial challenge. It goes well beyond our current paradigms for building *secure* systems because it calls for consideration of safety aspects and anticipation of threats beyond the foreseeable threat horizon. It also goes beyond our current thinking about *safety* by broadening the scope to include deliberate faults induced by an adversary. Nevertheless, sustainable security & safety will become increasingly necessary as we become increasingly dependent on these (ICT) systems. Even if the full vision of sustainable security & safety is not fully achieved, any advances in this direction could have a significant impact on the design of future system. We have set out our vision for sustainable security & safety, identified the main challenges currently faced, and proposed a set of design principles towards overcoming these challenges and achieving this vision.

References

1 Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. on Information System Security*, 13, 2009.

2 National Highway Traffic Safety Administration. Vehicle Survivability and Travel Mileage Schedules, 2006. URL: `https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/809952`.

3 Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January 2004. `doi:10.1109/TDSC.2004.2`.

4 Alysson Neves Bessani, Paulo Sousa, Miguel Correia, Nuno Ferreia Neves, and Paulo Verissimo. The Crutial Way of Critical Infrastructure Protection. *IEEE Security Privacy*, 6(6):44–51, November 2008. `doi:10.1109/MSP.2008.158`.

5 Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *24th USENIX Security Symposium*, USENIX Sec, 2015.

6 ENISA. Algorithms, Key Sizes and Parameters Report - 2013 recommendations. Technical report, www.enisa.europe.eu, October 2013.

7 Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Annual Network & Distributed System Security Symposium (NDSS)*, 2008.

8 Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Queue*, 10(1), January 2012.

9 ISO Technical Committee 22/SC 32. *ISO26262: Road vehicles - Functional safety*, 2018.

10 Antonio Lima, Francisco Rocha, Marcus Völp, and Paulo Esteves-Veríssimo. Towards Safe and Secure Autonomous and Cooperative Vehicle Ecosystems. In *2ndACM Workshop on Cyber-Physical Systems Security and Privacy (co-located with CCS)*, Vienna, Austria, October 2016.

11 Rowan McAllister, Yarin Gal, Alex Kendall, Mark Van Der Wilk, Amar Shah, Roberto Cipolla, and Adrian Weller. Concrete Problems for Autonomous Vehicle Safety: Advantages of Bayesian Deep Learning. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI, 2017.

12 Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33(12), December 1990.

13 Peter Oehlert. Violating Assumptions with Fuzzing. *IEEE S&P*, 3(2), March 2005.

14 Andrew Paverd, Marcus Völp, Ferdinand Brasser, Matthias Schunter, N. Asokan, Ahmad-Reza Sadeghi, Paulo Esteves Verissimo, Andreas Steininger, and Thorsten Holz. Sustainable Security & Safety: Challenges and Opportunities. long version avail. at: `http://www.icri-cars.org/wp-content/uploads/2019/01/s3-vision.pdf`, May 2019.

15 Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *36th IEEE Symp. on Security and Privacy*, 2015.

16 Adi Shamir. How to Share a Secret. *Commun. ACM*, 22(11):612–613, November 1979. `doi:10.1145/359168.359176`.

17 Kevin Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *34th IEEE Symposium on Security and Privacy (Oakland 2013)*, 2013.

18 Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreia Neves, and Paulo Verissimo. Highly Available Intrusion-Tolerant Services with Proactive-Reactive Recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, April 2010. `doi:10.1109/TPDS.2009.83`.

19 Shruti Tople, Karan Grover, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and Secure DNN Inference. *CoRR*, abs/1810.00602, 2018. `arXiv:1810.00602`.

20 Paulo Verissimo, Miguel Correia, Nuno Ferreira Neves, and Paulo Sousa. Intrusion-Resilient Middleware Design and Validation. In *Information Assurance, Security and Privacy Services*, volume 4 of *Handbooks in Information Systems*, pages 615–678. Emerald Group Publishing Limited, May 2009. URL: `http://www.navigators.di.fc.ul.pt/archive/papers/annals-IntTol-compacto.pdf`.

21 Paulo Verissimo, Nuno Ferreira Neves, and Miguel Correia. Intrusion-Tolerant Architectures: Concepts and Design. In *Architecting Dependable Systems*, volume 2677 of *LNCS*, pages 3–36. Springer-Verlag, June 2003. Extended version in http://hdl.handle.net/10455/2954. URL: `http://www.navigators.di.fc.ul.pt/docs/abstracts/archit-03.html`.

22 Marcus Völp, Francisco Rocha, Jeremie Decouchant, Jiangshan Yu, and Paulo Esteves-Verissimo. Permanent Reencryption: How to Survive Generations of Cryptanalysts to Come. In Frank Stajano, Jonathan Anderson, Bruce Christianson, and Vashek Matyáš, editors, *Security Protocols XXV*, pages 232–237, Cham, 2017. Springer International Publishing.

On Fault-Tolerant Scheduling of Time Sensitive Networks

Radu Dobrin
Mälardalen University, Västerås, Sweden
radu.dobrin@mdh.se

Nitin Desai
Mälardalen University, Västerås, Sweden
nitin.desai@mdh.se

Sasikumar Punnekkat
Mälardalen University, Västerås, Sweden
sasikumar.punnekkat@mdh.se

Abstract

Time sensitive networking (TSN) is gaining attention in industrial automation networks since it brings essential real-time capabilities at the data link layer. Though it can provide deterministic latency under error free conditions, TSN still largely depends on space redundancy for improved reliability. In many scenarios, time redundancy could be an adequate as well as cost efficient alternative. Time redundancy in turn will have implications due to the need for over-provisions needed for timeliness guarantees. In this paper, we discuss how to embed fault-tolerance capability into TSN schedules and describe our approach using a simple example.

2012 ACM Subject Classification Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Time sensitive networks(TSN), Fault-tolerant schedule, Time redundancy

Digital Object Identifier 10.4230/OASIcs.CERTS.2019.5

Funding The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 764785, FORA—Fog Computing for Robotics and Industrial Automation.

1 Introduction

Time- and safety-critical control applications in the context of factories need real-time guarantees [26]. Commonly, such requirements are specified at design time and the system is expected to fulfil them during its entire operational life. This necessitates *guaranteed* and *bounded* latencies and low jitter for tasks/functions that are critical to safety (for the end-user) [24]. The design and development of distributed embedded systems driven by the Time-Triggered paradigm [17] has proven effective in a diversity of domains with stringent demands of determinism [7].

There has been a steady evolution from centralized control with the control logic embedded within a single controller to decentralized/distributed control where control is shared between multiple controllers. A key benefit of this is to provide greater robustness to failures. For instance, a distributed architecture is more conducive to safety, by ensuring critical functions have the possibility of being executed at multiple physical nodes and transported across multiple communication links (the basic notion of redundancy). However, from a network latency perspective, this may cause additional latencies due to multiple hops (when an alternate link or node is needed). When timeliness is of the essence, such an arrangement may not therefore be optimal in providing determinism.

In applications such as factory automation or automobiles, the systems could be subjected to high degrees of Electro Magnetic Interference (EMI) from the operational environment which can cause transmission errors. The common causes for such interference include cellular phones and other radio equipment inside the premises/vehicle, electrical devices like switches and relays, radio transmissions from external sources and lightning in the environment. Complete elimination of the effects of EMI is hard since exact characterization of all such interference defy comprehension. Though usage of an all-optical network could greatly eliminate EMI problems, it may not be favoured by many cost-conscious industries.

These interferences cause errors in the transmitted data, which could indirectly lead to catastrophic failures. To reduce the risks due to erroneous transmissions, designers usually provide elaborate error checking and error confinement features in the protocol (as in Controller Area Networks). Basic philosophy of these features is to identify an error as fast as possible and then re-transmit the affected message. This implies that in systems without spatial redundancy of communication medium/controllers, the fault-tolerance (FT) mechanism employed is time redundancy. On the other hand, time redundancy increases the latency of message sets; potentially leading to violation of timing requirements. Hence any reliability management approaches in critical systems needs to be a holistic one incorporating both space and time redundancy at the right levels based on the system characteristics, resource constraints, fault models and trade-offs from cost-performance perspectives.

The time sensitive networking (TSN) [11] is a set of evolving standards under the IEEE working group IEEE802.1, defining protocols that extend standard Ethernet to achieve real-time networking capabilities for industrial/factory automation application scenarios. The TSN standardization efforts consists of a number of (sub)standards that aim to achieve four key technological paradigms - clock synchronization (802.1ASrev), frame preemption (802.1Qbu), scheduled traffic (802.1Qbv), and redundancy management (802.1CB). These must work together at the Ethernet layer (L2) to ensure that safety functions are executed while meeting their respective deadlines and constraints.

The 802.1Qbv TSN standard provides scheduled traffic for time-triggered safety-critical data frames in a predetermined manner. However, in the presence of faults, a static schedule cannot satisfy system requirements particularly since the schedule has to be reconfigured.

Redundancy management in TSN (802.1CB) has been mainly focussing on space (link redundancy). It is typical to start with a simplified error model assumption that only singleton errors can occur in the systems and that they are separated at least by a known minimum interarrival time.

In this paper, our focus is on time redundancy and how to improve fault tolerance capability of the TSN schedule. The underlying assumptions and models in our work are in line with our previous work and also with that followed in [2] [1]. We extend our earlier works presented in [10] and adapt it to the context of TSN.

The rest of the paper is organised as follows. Section 2 introduces the reader to the concept of time sensitive networking. Section 3 describes our system model and outlines its basic components. In section 4 we detail the proposed approach. In section 5, we present an illustrative example for our approach. Section 6 discusses the research relevant and related to our work and finally we conclude with section 7 and briefly sketch our ongoing efforts.

2 Time Sensitive Networking

The TSN standard is composed of a number of sub-standards. The most relevant sub-standard for our use-case is 802.1Qbv - *scheduled traffic*. In this use-case, we assume that the 802.1Qbv protocol is implemented both in the TSN switch as well as the control nodes and robots. A fundamental principle behind TSN [11] is the time-triggered protocol (TTP)

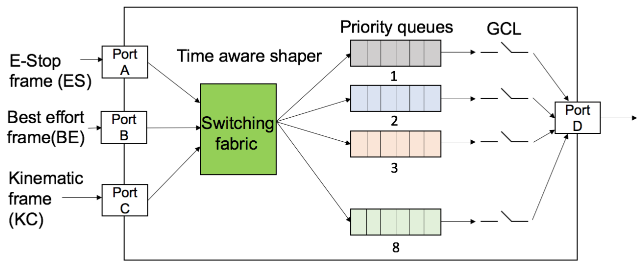

Figure 1 TSN 802.1Qbv-enabled switch.

[18]. In Figure 1, we have three ingress ports A, B, and C and a single egress port D. The safety frames (ES, KC) from control nodes are sent from ports A and B while port C sends best effort (BE) frames as shown. The Gate Control List (GCL) decides the exact times when the frames belonging to a specific priority queues will be allowed to pass through the egress port D. From a system safety perspective, ES and KC frames must be given higher priority than BE frames.

Such systems depend on redundant communication schedules that contain global time-based information of message transmissions with conflict-free paths through the switches. The static schedule of a time-triggered system maximizes predictability, while the schedule in an event-triggered network unfolds dynamically at run-time depending on the occurrence of events [18]. A time-triggered network ensures the partitioning of the system into a set of independent fault containment regions (FCR), which operate correctly regardless of an arbitrary fault outside the region.

3 System model

Having provided the background required for this use-case, the problem we tackle can be stated as follows:

"How do you guarantee delivery of safety-critical data frames across a TSN enabled network in the presence of faults specified by a fault model?"

In order to quantify relevant system parameters, we present a system model that is composed of sub-models that tackles each aspect of the system function.

3.1 System and error model

We assume a distributed real-time architecture consisting of sensors, actuators and processing nodes communicating over a time sensitive network. The communication is performed via a set of strict periodic messages, $\Gamma = \{M_1, M_2, \ldots\}$, with mixed criticality levels. The criticality of a message indicates the severity of the consequences caused by its failure and corresponds to the amount of resources allocated for error recovery in terms of guaranteed re-transmissions. The basic assumption here is that the effects of a large variety of transient and intermittent (hardware) faults can effectively be tolerated by a simple re-transmission of the affected frames. We assume that a fault can adversely affect only one message frame at a time and is detected by all nodes in the network. Γ_c represents the subset of critical messages out of the original message set and Γ_{nc} represents the subset of non-critical messages, so that $\Gamma = \Gamma_c \cup \Gamma_{nc}$.

A message consists of N frames, $N \geq 1$, and the network communication is assumed to be non-preemptive during the frame transmissions. Though sub-standard 802.1Qbu is introducing preemption of frames in TSN, for simplicity's sake we have not considered it in current work. Of course, messages composed of more than 2 frames can preempt each other at frame boundaries. Additionally, the non-preemptiveness of message frames may cause a higher priority message to be blocked by a lower priority message on the same link for at most one frame length. This priority inversion phenomenon can affect all messages except the lowest priority one, and only once per message period, before the transmission of the first message frame [9].

Each message M_i is characterized by a 4-tuple $< T_i, D_i, N_i, R_i >$, where T_i is the period, D_i is the relative deadline, N_i is the number of frames that form this message and R_i is the fault tolerant requirement in terms of the number of re-transmissions the message needs to be able to execute upon faults. Hence, the total number of frames that need to be guaranteed for re-transmission r_i is calculated by

$$r_i = \lceil N_i * R_i \rceil \qquad (1)$$

Note that for non-critical messages $R_i = 0$. Additionally, rate constrained and best effort messages have a priority P_i.

In an error-free scenario, the worst case transmission time C_i of message M_i is

$$C_i = N_i * f * \tau_{bit} \qquad (2)$$

where f is the maximum frame size and τ_{bit} is the transmission time for a bit.

Each *message instance* M_i^j is characterized by a *feasibility window* delimited by its earliest start time $est(M_i^j)$ and its deadline D_i^j.

Obviously, in order to be able to guarantee the specified fault tolerance requirements, the maximum network utilization of the critical messages together with their required re-transmissions can never exceed 100% of the bandwidth capacity. This will imply that, during the error recovery, non-critical message transmissions may need to be shed in order to avoid overload conditions.

3.2 Traffic model

Real-time traffic in control systems is highly regular and periodic. The schedules for such traffic can be statically synthesized during design phase. This plan may not only define the communication paths and bandwidth reservations, but also particular points in a network-wide reference time at which messages are to be transmitted. Such a plan that incorporates the time aspects is called a "communication schedule" and the execution of the schedule by the network obeys the time-triggered approach. Safety critical messages are usually transmitted through time-triggered (TT) class since bounded delivery latency is guaranteed [24].

A basic fault tolerance mechanism in the presence of faults is to re-transmit an alternate of the original message at a later time instant. This is suitable for single errors during message transmissions. It is also assumed that no errors affect the alternate message transmission. For simple cases one could consider the re-transmission of the original message itself, but the approach could as well cater to initiation of another alternate task leading to an alternate message (for example in critical scenarios warranting an "emergency stop").

4 Proposed approach

Our research objective is to provide efficient and fault-tolerant scheduling algorithms and mechanisms for TSN that ensure:

1. All safety critical messages (time triggered traffic) have guaranteed correct delivery within their deadlines under given fault assumptions
2. All non-critical traffic is given best-effort schedulability guarantees
3. The generated schedules also possess flexibility to incorporate evolving changes traffic patterns particularly in the absence of faults

We make the following assumptions to start with:

1. A fault can affect only one message at a time
2. A specified number of re-transmissions of the message is sufficient to overcome the effect of a transient or intermittent fault.
3. There exist sufficient fault detection capabilities (such as watchdog timers, CRC etc.) in the system so that a fault can be detected reliably within a specified short time interval.
4. There exist ARQ mechanisms so that the sender node is able to know within a specified time whether the message sent has reached the destination (or intermediate node) correctly.

Our ongoing research efforts aims at providing specific contributions in the following directions:

1. The use of phased re-transmissions that can achieve better bandwidth utilization than possible with the approach of Alvarez et. al. [1].
2. Combined scheduling of critical and non-critical messages using the concept of fault-tolerant windows and fault aware windows
3. Making more realistic fault model assumptions
4. Making our schemes more flexible to support evolution of systems
5. Suggesting mechanisms for implementation for induction into standards

The focus of current paper is only first two items above. Here we propose an approach to jointly schedule critical and non-critical messages as time triggered and rate constrained traffic in TSN. We propose to schedule critical messages with completely known attributes as time triggered traffic. Some critical messages, however, may have requirements that cannot be accommodated in an off-line schedule a-priori. These are, instead, scheduled as rate constrained (RC). It is essential, however, to schedule them at priority levels that guarantee their re-transmissions in case of faults. At the same time, we aim to provide the non-critical best effort traffic the best possible service in case the system is not overloaded due to faults.

The key concept in our proposed approach is the derivation of the feasibility windows for the message transmissions. Traditionally the feasibility window for a message is the time interval between its earliest start time (or release time) and its deadline. These parameters, however, do not typically express the fault tolerant requirements on the critical messages, e.g., a message transmission finishes just before its deadline, will not leave enough time for a feasible (before its deadline) re-transmission in case the message is hit by a fault. We propose the derivation of new feasibility windows for each message instance $M_i^j \in \Gamma$ that reflect the FT requirements.

While transmitting non-critical messages using a background priority band can be a safe and straightforward solution, our aim is to provide non-critical messages a better service than what can be achieved through background scheduling. Hence, depending on the criticality of the original set of messages, the new feasibility windows we are looking for differ as:

1. *Fault-Tolerant* (FT) feasibility windows for critical messages
2. *Fault-Aware* (FA) feasibility windows for non-critical messages

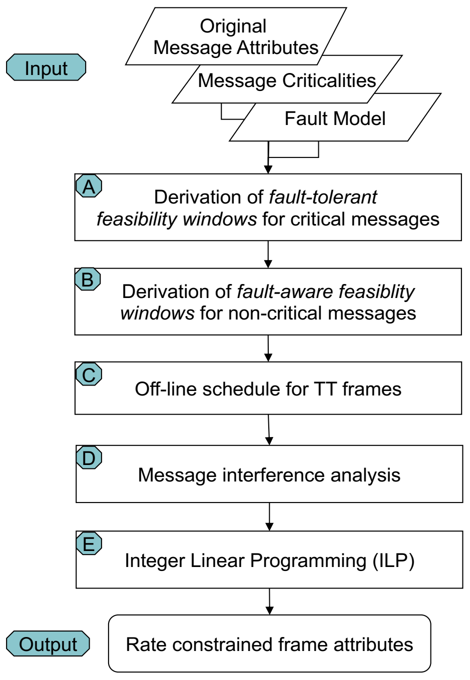
Input
Original Message Attributes
Message Criticalities
Fault Model
A
Derivation of *fault-tolerant feasibility windows* for critical messages
B
Derivation of *fault-aware feasiblity windows* for non-critical messages
C
Off-line schedule for TT frames
D
Message interference analysis
E
Integer Linear Programming (ILP)
Output
Rate constrained frame attributes

Figure 2 Methodology overview.

While critical messages need to be entirely transmitted within their FT feasibility windows to be able to be feasibly re-transmitted upon an error, according to the reliability requirements, the derivation of FA feasibility windows has two purposes: 1) to prevent non-critical messages from interfering with critical ones thus causing a critical message to miss its deadline, while 2) enabling the transmission of the non-critical messages at high priority levels in error free situations.

The major steps of the proposed methodology are shown in Figure 2. The inputs to the method are message attributes, criticalities and fault model in terms of frequency of faults and fault-tolerance requirements.

Since the size of the FA feasibility windows depends on the size of the FT feasibility windows, in our approach we first derive FT-feasibility windows and then FA feasibility windows (as steps A and B Figure 2). Then, we assign time slots for TT traffic and priorities to rate constrained traffic to ensure the message transmissions within their newly derived feasibility windows.

Subsequently we generate an off-line schedule for the TT traffic (in step C) followed by assigning message identifiers (priorities) for the rate constrained traffic (in step D) that ensure the message transmissions within their new feasibility windows, thus, fulfilling the FT requirements. We generate an offline schedule for the TT messages, by using the Earliest Deadline First (EDF) heuristics and provisioning for the specified number of re-transmission upon faults. Then we identify the optimal priorities for the critical rate based traffic in order to ensure its coexistence with the TT traffic, as well as its FT requirements. At the same time, we derive the priorities for the non-critical messages that ensure their timeliness in the absence of faults. As the network utilisation will heavily increase due to the re-transmissions of the critical messages under faults, we assume that in these situations the non-critical messages are shed by their sending nodes.

In some cases, however, a fixed priority scheme cannot express all our assumed FT requirements and error assumptions on the rate constrained traffic. General FT requirements may require that instances of a given set of periodic messages needs to be transmitted in different order on different occasions. Obviously, there exists no valid fixed priority assignment that can achieve these different orders. Our approach proposes a priority allocation scheme based on EDF at message instance level that efficiently utilizes the resources while minimizing the priority levels. We use Integer Linear Programming (ILP) (Step E) to off-line analyze the interference between the message frames and to derive the minimum number of fixed priorities that guarantees the message transmissions within their FT/FA Feasibility Windows.

5 Discussions

We discuss our approach by resorting to a simple but instructive example detailed below.

A set of three messages A,B, and C are considered, wherein A and C are critical and B is non-critical with periods $T(A) = 4$, $T(B) = 8$, $T(C) = 16$ and transmission times, $C(A) = 1$, $C(B) = 4$, $C(C) = 3$. We assume the deadlines for the messages equal their periods. We re-transmit only critical messages when subject to a single fault per message instance. Figure 3 shows a feasible message transmission under the assumption of "no faults".

Our proposed approach is illustrated in a set of figures depicting various scenarios and schedules. As part of our motivation for the proposed fault tolerant windows based approach, we first show the Rate monotonic(RM) schedule for the message transmissions in Figure 3.

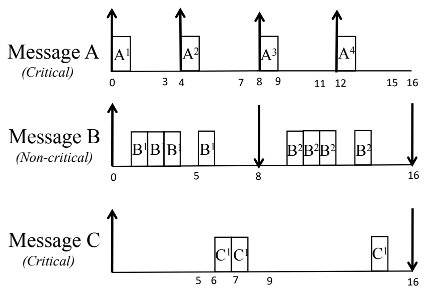

Figure 3 RM-based schedule with no faults - but not an FT schedule.

Figure 4 shows the infeasibility of the critical message C in case 2 instances of A a hit by faults and need to be re-transmitted.

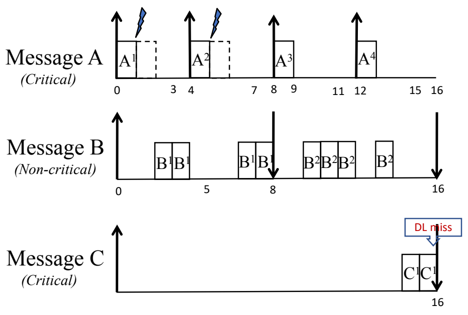

Figure 4 Two faults on message A causing even primary of critical message C to miss deadline.

Figure 5 shows that that the critical message C cannot even tolerate a single fault.

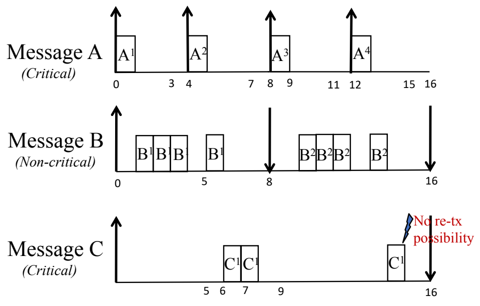

Figure 5 A single fault in message C prevents re-transmission possibilities.

A solution could be, however, to increase the priority of the critical message C above the priority of the non-critical message B. In this case, however, the first instance of B will always miss its deadline, even in a fault free scenario (Figure 6).

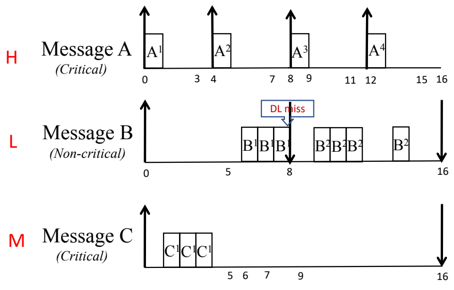

Figure 6 Priority modification (non-RM) still causing B to miss deadline.

Figure 7 illustrates the derivation of the fault tolerant (FT) windows for critical messages A and C. The dashed boxes represent the re-transmissions that would be needed if the critical messages were to experience a single fault per instance.

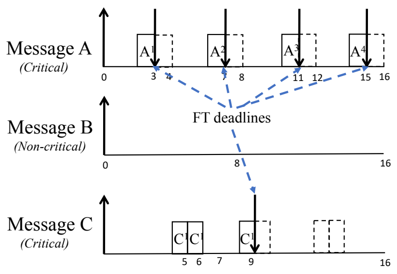

Figure 7 Derivation of FT windows for critical messages.

Figure 8 shows the derivation of fault aware (FA) windows for non-critical message (B). This is done after the fault tolerant windows for the critical messages have been calculated. Figure 9 shows a resulting schedule which would meet all deadlines under a fault free scenario.

Figure 10 illustrates the benefits provided by our approach. The critical message A is transmitted in a time-triggered (TT) manner while the other messages are assigned to the rate-constrained traffic class (RC). We have three faults occurring on the critical messages. Our scheduling approach ensures that all critical messages are scheduled in a fault tolerant manner while only one instance of the non-critical message fails to meet the deadline.

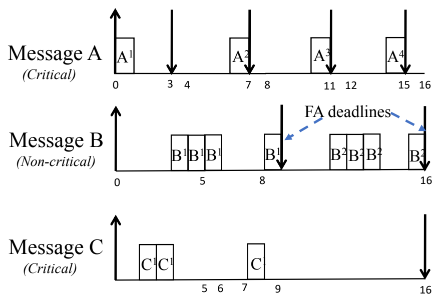

Figure 8 Derivation of fault-aware windows for non-critical messages.

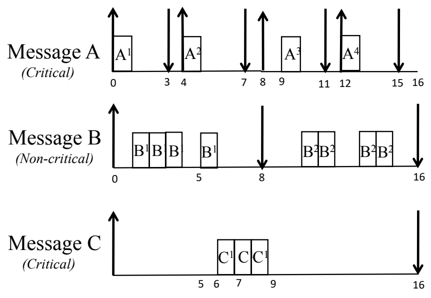

Figure 9 Fault free messages with no deadline misses.

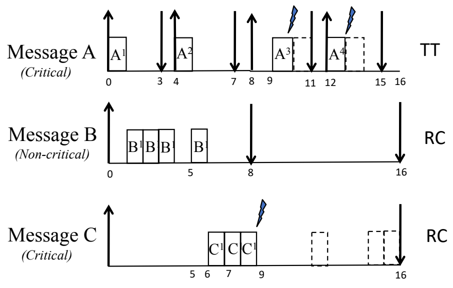

Figure 10 Three faults with all critical messages scheduled and only one instance of non-critical message un-schedulable.

Finally, Figure 11 shows a scenario with 5 faults. Critical messages A and C still remain fault tolerant while the non-critical message B is prevented from execution. However, in the event that critical messages do not experience faults, the non-critical message B can still meet its deadline, thereby providing a better service than background scheduling.

In summary, the above example scenarios shows that:

- For a given message set A,B,C where A and C are critical, a RM priority assignment (A>B>C) will not guarantee the 100% FT (i.e. one re-transmission upon a potential fault per message instance). C will miss its deadline.
- A new priority ordering where the non-critical message has a lower priority than the critical message (A>C>B) will guarantee the 100% FT of the critical messages but B will miss its first deadline even in a fault free scenario

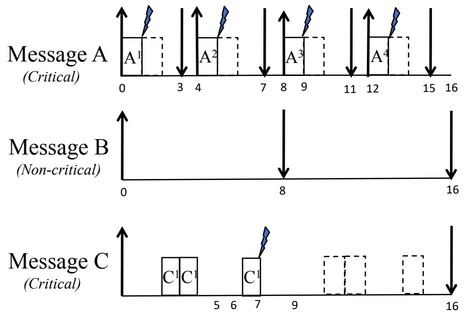

Figure 11 Each critical message instance hit by a fault can be re-transmitted within its deadline.

What we propose is a new stream/message allocation and priority assignment that:

- Maximises the FT capability of critical messages in the presence of fault
- Maximises the service of the non-critical messages in the absence of faults

We derive FT feasibility windows and FA feasibility windows based on Chetto and Chetto [6] and further Aysan et al [4] that ensure the above. In the example, we put A in the TT traffic and B and C in the RC traffic with priorities derived by an ILP solver given the feasibility windows constraint.

6 Related work

For scheduling on time-triggered networks, Steiner [22] introduced a method to synthesize the time-triggered traffic using an SMT YICES solver. A common approach is to have a "recovery slack" in the schedule in order to accommodate time needed for re-executions in case of faults [10]. It has been shown that the time-triggered paradigm which forms a core part of the time sensitive network standard (as 802.1Qbv time-aware shaper [23]) ensures the *fail-silent semantics* whereby a packet is received only if correct or not received at all. Fault recovery studies such as [25] depend on a fast spanning tree reconfiguration algorithm to reduce the total fault recovery time, and a delayed link inactivation scheme that allows real-time connections which are not affected by the failed links/switches to continue to exist.

Recent approaches by Steiner et al [21][13] based on reconfiguration of GCL schedules at runtime for 802.1Qbv TSN discuss a configuration agent that is aware of the traffic conditions at each node in the network. The objective is to ensure that new traffic flows can be accommodated with use of as few queues in the switch ports as possible while maintaining a feasible schedule.

Proenza et al [5] [19] have proposed a *flexible* time triggered paradigm for distributed real time systems. Flexibility refers to the adaptation of the nodes to new and evolving hard real-time requirements such as periodic and sporadic messages and updating the parameters of such messages at run-time. Some recent studies on reconfiguration by means of spatial and temporal techniques are discussed in [2] [3]. Desai et al [8] discuss safety of industrial automation systems, although focused towards fog/edge paradigms. Pozo et al [20] have shown that schedules can be "repaired" to combat the presence of faults.

Recovery from a transient or permanent fault in a time triggered network implies a certain amount of flexibility in the mechanisms to ask for changes in real-time requirements at runtime to reconfigure nodes and switches according to a new schedule [12]. Gutiérrez et al [14] and Raagard et al [21] discuss a configuration agent that can synthesize new schedules for TSN at runtime. Time synchronization aspects are also extremely crucial and addressed in works such as [16] and [15].

7 Conclusion and Ongoing Work

Scheduling of safety-critical data frames (and tasks) constitutes a fundamental design requirement. The principal limitation of the time-triggered approach is the inability to adapt to unanticipated changes in the system parameters such as traffic patterns or faults. This causes the schedule not to guarantee the transmission of all frames within their timing requirements. If the network does not contain a backup schedule predicting that specific change, the schedule needs to be synthesized again from scratch, which is computationally and time intensive.

With respect to 802.1Qbv, our goal is to ensure that the schedule offsets representing the opening and closing of the gates (the GCL table) for the TSN switches are recalculated first for the TT traffic while simultaneously meeting the timing requirements (deadlines) for message transmissions.

In this paper we saw how our FT/FA aware scheduling approach provides critical messages to meet their deadlines even when all instances of the critical messages experience single faults. Additionally, in case faults do not occur, we have the possibility for non-critical messages to be served in a better way (compared to background scheduling). We are currently in the process of performing detailed evaluation of the approach thorough simulations.

As part of ongoing work, we are focusing on transmitting critical messages as time-triggered traffic which will enforce a much stricter time assignment.

References

1 I. Álvarez, C. Drago, J. Proenza, and M. Barranco. First Study of the Proactive Transmission of Replicated Frames Mechanism over TSN. *16th International Workshop on Real Time Networks (RTN), ECRTS*, 2018.

2 I. Álvarez, J. Proenza, and M. Barranco. Mixing Time and Spatial Redundancy Over Time Sensitive Networking. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, June 2018. `doi:10.1109/DSN-W.2018.00031`.

3 M. Ashjaei, P. Pedreiras, M. Behnam, L. Almeida, and T. Nolte. Evaluation of dynamic reconfiguration architecture in multi-hop switched ethernet networks. In *IEEE Emerging Technology and Factory Automation*, pages 1–4, September 2014. `doi:10.1109/ETFA.2014.7005322`.

4 H. Aysan, R. Dobrin, and S. Punnekkat. Fault Tolerant Scheduling on Control Area Network (CAN). *IEEE International Workshop on Object/component/service-oriented Real-time Networked Ultra-dependable Systems*, 2010.

5 A. Ballesteros, J. Proenza, and P. Palmer. Towards a dynamic task allocation scheme for highly-reliable adaptive distributed embedded systems. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4, September 2017. `doi:10.1109/ETFA.2017.8247773`.

6 H. Chetto and M. Chetto. Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Transactions on Software Engineering*, 15(10), October 1989. `doi:10.1109/TSE.1989.559777`.

7 S.S. Craciunas and R.S. Oliver. Combined Task- and Network-level Scheduling for Distributed Time-triggered Systems. *Real-Time Systems Journal*, 52(2), March 2016.

8 N. Desai and S. Punnekkat. Safety of Fog-based Industrial Automation Systems. In *Proceedings of the Workshop on Fog Computing and the IoT*. ACM, 2019. `doi:10.1145/3313150.3313218`.

9 M. Di Natale. Scheduling the CAN Bus with Earliest Deadline Techniques. *IEEE Real-Time Systems Symposium*, pages 259–268, November 2000.

10 R. Dobrin, H. Aysan, and S. Punnekkat. Maximizing the Fault Tolerance Capability of Fixed Priority Schedules. In *RTCSA*, pages 337–346, September 2008. `doi:10.1109/RTCSA.2008.6`.

11 N. Finn. Introduction to Time-Sensitive Networking. *IEEE Communications Standards Magazine*, 2(2):22–28, June 2018. `doi:10.1109/MCOMSTD.2018.1700076`.

12 D. Gessner, J. Proenza, M. Barranco, and A. Ballesteros. A Fault-Tolerant Ethernet for Hard Real-Time Adaptive Systems. *IEEE Transactions on Industrial Informatics*, pages 1–1, 2019. `doi:10.1109/TII.2019.2895046`.

13 M. Gutiérrez, A. Ademaj, W. Steiner, R. Dobrin, and S. Punnekkat. Self-configuration of IEEE 802.1 TSN networks. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, September 2017. `doi:10.1109/ETFA.2017.8247597`.

14 M. Gutiérrez, W. Steiner, R. Dobrin, and S. Punnekkat. A configuration agent based on the time-triggered paradigm for real-time networks. In *2015 IEEE World Conference on Factory Communication Systems (WFCS)*, pages 1–4, May 2015. `doi:10.1109/WFCS.2015.7160584`.

15 Marina Gutiérrez, Wilfried Steiner, Radu Dobrin, and Sasikumar Punnekkat. Synchronization Quality of IEEE 802.1AS in Large-Scale Industrial Automation Networks. In *23rd IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2017.

16 E. Heidinger, F. Geyer, S. Schneele, and M. Paulitsch. A performance study of Audio Video Bridging in aeronautic Ethernet networks. In *IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, pages 67–75, June 2012. `doi:10.1109/SIES.2012.6356571`.

17 H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, January 2003. `doi:10.1109/JPROC.2002.805821`.

18 H. Kopetz and G. Grunsteidl. TTP - a time-triggered protocol for fault-tolerant real-time systems. In *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, pages 524–533, June 1993. `doi:10.1109/FTCS.1993.627355`.

19 P. Pedreiras and A. Luis. The flexible time-triggered (FTT) paradigm: an approach to QoS management in distributed real-time systems. In *Proceedings International Parallel and Distributed Processing Symposium*, April 2003. `doi:10.1109/IPDPS.2003.1213243`.

20 F. Pozo, G. Rodriguez-Navas, and H. Hansson. Schedule Reparability: Enhancing Time-Triggered Network Recovery Upon Link Failures. In *RTCSA*, 2018. `doi:10.1109/RTCSA.2018.00026`.

21 M. L. Raagaard, P. Pop, M. Gutiérrez, and W. Steiner. Runtime reconfiguration of time-sensitive networking (TSN) schedules for Fog Computing. In *2017 IEEE Fog World Congress (FWC)*, pages 1–6, October 2017. `doi:10.1109/FWC.2017.8368523`.

22 W. Steiner. An Evaluation of SMT-Based Schedule Synthesis for Time-Triggered Multi-hop Networks. In *IEEE Real-Time Systems Symposium*, November 2010. `doi:10.1109/RTSS.2010.25`.

23 W. Steiner, S. S. Craciunas, and R. S. Oliver. Traffic Planning for Time-Sensitive Communication. *IEEE Communications Standards Magazine*, 2(2):42–47, June 2018. `doi:10.1109/MCOMSTD.2018.1700055`.

24 D. Tamas-Selicean, P. Pop, and W. Steiner. Synthesis of Communication Schedules for TTEthernet-based Mixed-criticality Systems. In *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2012. `doi:10.1145/2380445.2380518`.

25 S. Varadarajan and T. Chiueh. Automatic fault detection and recovery in real time switched Ethernet networks. In *IEEE INFOCOM Conference on Computer Communications*, March 1999. `doi:10.1109/INFCOM.1999.749264`.

26 M. Wollschlaeger, T. Sauter, and J. Jasperneite. The Future of Industrial Communication: Automation Networks in the Era of the Internet of Things and Industry 4.0. *IEEE Industrial Electronics Magazine*, 11(1):17–27, March 2017. `doi:10.1109/MIE.2017.2649104`.